Exemples de developpement de programmes dans la theorie des constructions

C. Mohring

INRIA
ENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. : (1) 39 63 55 11

Rapports de Recherche

N° 497

EXEMPLES DE DÉVELOPPEMENT DE PROGRAMMES DANS LA THÉORIE DES CONSTRUCTIONS

Christine MOHRING

Mars 1986

EXEMPLES DE DEVELOPPEMENT DE PROGRAMMES DANS LA THEORIE DES CONSTRUCTIONS

Christine MOHRING

Résumé

Nous présentons le développement et la preuve d'algorithmes écrits dans le formalisme des constructions de G. Huet et Th. Coquand. Les programmes sont des λ-termes typés, dont on peut extraire un λ-terme pur qui représente la partie éxecutable de l'algorithme. La vérification des types et donc la preuve que le programme obéit à sa spécification, est automatiquement vérifié par le système, une implémentation expérimentale de la théorie dans le langage ML. Nous présentons brièvement le calcul puis nous passons aux exemples (inverse d'une fonction entière, recherche d'un candidat majoritaire dans un vote, mise en page d'un texte et Quicksort), ceux-ci sont accompagnés de commentaires et de remarques.

Abstract

We present the development and proof of algorithms, written in the formalism of constructions of G.Huet and Th.Coquand. Programs are realized as typed λ-terms from where we can extract a pure λ-terme which represents the computable part of the algorithm. Type verification and therefore correctness of programs is mechanically checked by the system, a prototype implementation of the theory in the language ML. We briefly present this calculus and then the examples (inverse of a map from Nat to Nat, searching a majority in a vote, formatting and Quicksort), with remarks and comments.

Ce travail est l'extension d'un rapport de stage de DEA effectué sous la direction de Gérard Huet à l'INRIA de février à septembre 85.

Exemples de Développement de Programmes dans la Théorie des Constructions

Christine Mohring

INRIA
Domaine de Voluceau
78150 Rocquencourt
France

La théorie des constructions est un lambda calcul typé d'ordre supérieur développé par Thierry Coquand et Gérard Huet [5][6]. Un prototype d'implantation du système a été réalisé en ML à l'INRIA. Celui-ci permet de définir des constantes du langage et assure une vérification mécanique des types. Ainsi ont pu être effectués des codages et des preuves de résultats mathématiques [8].

Dans cette étude, nous avons utilisé le calcul des constructions pour développer des programmes conjointement à leur preuve. Ceci a été réalisé sur quatre examples; nous construisons un λ-terme dont le type correspond à la spécification du programme et dont on peut extraire l'algorithme cherché sous forme d'un λ-terme pur dont la réduction correspond à l'exécution du programme.

Nous commencerons par rappeler dans un premier paragraphe, les résultats à la base de cette théorie ainsi que les caractéristiques du langage des constructions. Les paragraphes suivants sont consacrés à des exemples de développement d'algorithmes : construction de la fonction *lambo*(f) inverse d'une fonction f sur les entiers, recherche d'un candidat majoritaire dans une liste de votes, mise en page ("formatting") et tri rapide (Quicksort). Nous donnerons l'écriture complète et commentée des programmes. Nous finirons par des remarques sur les difficultés rencontrées lors de ce travail.

1. Introduction

1.1. Des termes typés aux preuves et aux programmes.

Expliquons pourquoi un λ-calcul typé tel que le langage des constructions nous permet de coder des preuves et des programmes.

La possibilité d'écrire une preuve sous forme d'un λ-terme est justifiée par l'"isomorphisme de Curry-Howard", c'est à dire une analogie entre le lambda calcul typé et la logique des systèmes de déduction naturelle mise en évidence par Curry puis Howard [12] et De Bruijn [4]. Le tableau suivant donne des exemples de ces correspondances :

lambda calcul typé	logique
type	proposition
terme	preuve

normalisation	élimination des coupures
Fonction de A dans B: $\lambda x \in A \cdot B$	Implication intuitionniste: A⟶B
Identité: $\lambda x \in A \cdot x$	preuve de A⟶A
$\lambda x \in A. \lambda y \in B \cdot x$	preuve de A⟶(B⟶A)
produit cartésien de types : A×B	conjonction de propositions : A&B
première projection : A×B->A	règle de destruction : A&B->A
formation de la paire :A,B->A×B	règle d'introduction : A,B->A&B
produit dépendant: $\Pi_{x \in A} \cdot B(x)$	généralisation: $\forall x \in A \cdot B(x)$

Ces résultats sont mis en pratique dans le langage AUTOMATH développé et implémenté à Eindhoven sous la direction de N.G. De Bruijn [4]. Ce système permet le codage et la vérification mécanique de résultats mathématiques. Notons en particulier la transcription dans ce langage du livre de Landau : "Grundlagen der Analysis".

Des notions mathématiques ont été codées et prouvées dans la théorie des constructions [8][10]. Citons entre autres le lemme de Newman (toute relation noetherienne localement confluente est confluente) [8]. L'usage de types d'ordre supérieur permet d'écrire simplement une propriété telle que le caractère noethérien d'une relation. On remarque également les similitudes entre les preuves mécaniques et manuelles.

Des preuves aux programmes il n'y a qu'un pas. En effet écrire un algorithme n'est rien d'autre que faire une preuve constructive (sans raisonnement par l'absurde) d'une certaine proposition, à savoir la spécification du programme. On peut considérer un algorithme comme une fonction dont les arguments correspondent aux entrées et les valeurs aux sorties du programme. Construire une telle fonction, c'est prouver constructivement que pour chaque argument il existe une valeur ayant la propriété souhaitée. Par exemple tout algorithme de division euclidienne est une preuve de :

$$\forall n \in N, \forall q \in N^*, \exists p, r \in N \quad n = pq + r \text{ et } 0 \leq r < q$$

La différence fondamentale entre les points de vue informatique et mathématique est que le théorème sous-jacent à la plupart des spécifications ne présente pas de difficultés, par contre l'informaticien s'intéresse à la structure de la preuve qui représente son algorithme, pour en étudier par exemple la complexité. De plus cette preuve doit être assez formalisée pour être mécaniquement interprétée.

En résumé on peut voir les choses de trois manières [13]:

A est un type	A est une proposition	A est une tâche
a est de type A	a est une preuve de A	a est un programme pour la tâche A

Ces parallèles sont utilisés pour le développement d'algorithmes (division euclidienne, quicksort,..) dans le système de Martin Löf [16][17] ou dans d'autres systèmes [3][2].

La théorie des constructions offre une structure de types fondée sur un nombre réduit d'opérations mais d'une grande puissance d'expression. Elle permet une exploitation aisée et fructueuse des résultats précédents. L'utilisation de l'ordre supérieur nous permet de définir dans le langage les éléments de base: entiers, connecteurs...On verra que leur définition correspond à leur usage informatique en tant que structure de contrôle, en particulier les entiers sont des itérateurs, les booléens des opérateurs de branchement.

1.2. Présentation du langage des constructions

Dans sa thèse [6], Th. Coquand a étendu le formalisme du système AUTOMATH de N.G. De Bruijn [4] en un calcul qui contient les types du second ordre de J.Y Girard [11], ainsi qu'un produit dépendant qui, on le verra, nous permet d'exprimer les spécifications. Ce langage se rapproche également d'un ancien système de Martin-Löf. Ce calcul vérifie un théorème de normalisation (cf [6],[9] pour des démonstrations), il est en particulier consistent.

Des résultats formels plus détaillés sur la théorie des constructions se trouvent dans [5][6][7][8][9]. Notons que le calcul des constructions évolue actuellement tant sur le plan théorique (cf Coquand [6]) que sur le plan pratique (les nouvelles implantations sont plus interactives). Nous donnons ici un aperçu du calcul tel qu'il est présenté dans la thèse de Th. Coquand dans le but de permettre la lecture des exemples.

1.2.1. Termes du langage

Les termes utilisés sont certaines expressions bien formées du calcul Λ de Nederpelt [15]. Soit V un ensemble de variables, si M ∈ Λ alors V(M) désigne les variables libres du terme M. Λ est défini récursivement par :

$* \in \Lambda$	(constante représentant l'univers)
$V \subset \Lambda$	(variables)
$(M\ N) \in \Lambda$ si $M,N \in \Lambda$	(application)
$[x:M]N \in \Lambda$ si $M,N \in \Lambda$ et $x \in V\text{-}V(M)$	(abstraction)

La seule règle utilisée sur cet ensemble est la β-réduction :

$$(([x:A]B)M) \longrightarrow_\beta B[x/M]$$

où $B[x/M]$ représente le terme B dans lequel M est substitué aux occurrences libres de la variable x.

On montre que Λ vérifie la propriété de *Church-Rosser* pour cette règle, ceci permet entre autre de définir la notion de β-conversion notée $=_\beta$ et définie par :

$$M =_\beta N \Leftrightarrow \exists P \text{ tel que } M \rightarrow_\beta P \text{ et } N \rightarrow_\beta P$$

1.2.2. Expressions bien formées

On va montrer comment construire une expression $B \mid\!- E$. B est appelé environnement, il est soit vide, soit de la forme : $x_1:A_1,\ldots,x_n:A_n$ (on note alors B_{x_i} le terme A_i). E est soit un terme C de Λ soit de la forme $M \in N$ avec M et N dans Λ.

Définition : Une construction est une expression $B \mid\!- E$ dont il existe une dérivation par les règles ci-dessous :

Environnements

$\mid\!- *$

si $B \mid\!- C$ et x ne figure pas dans B alors $B,x:C \mid\!- *$

si $B \mid\!- P \in *$ et x ne figure pas dans B alors $B,x:P \mid\!- *$

Introduction de variables

si $B \mid\!- *$ et x figure dans B alors $B \mid\!- x \in B_x$

Lambda-introduction (abstraction et produit)

si $B,x:N \mid\!- M \in L$ alors $B \mid\!- [x:N]M \in [x:N]L$

si $B,x:N \mid\!- C$ alors $B \mid\!- [x:N]C$

si $B,x:N \mid\!- P \in *$ alors $B \mid\!- [x:N]P \in *$

Application

si $B \mid\!- M_0 \in N$ et $B \mid\!- M_1 \in [x:N]L$ alors $B \mid\!- (M_1 M_0) \in L[x / M_0]$

Egalité des types

Si $B \mid\!- P \in C$ et $B \mid\!- C'$ et $C =_\beta C'$ alors $B \mid\!- P \in C'$

si $B \mid\!- M \in P$ et $B \mid\!- P' \in *$ et $P =_\beta P'$ alors $B \mid\!- M \in P'$.

On dit que C est un ***contexte***, s'il existe une dérivation de $B \mid\!- C$. Les contextes sont les seules expressions non typées de la théorie, ils ont pour forme $[x_1:A_1]..[x_n:A_n]*$ ou $*$.

On dit que P est une ***proposition***, s'il existe une dérivation de $B \mid\!- P \in C$, où C est un contexte. On remarque que si $B \mid\!- P \in [x:A]*$ alors il existe une dérivation de $B \mid\!- P \in *$. Ceci est du au fait que les crochets d'abstraction représentent à la fois les produits et les fonctions (à la manière d'Automath). P est une proposition dépendante d'un élément de A ou la proposition $\forall x \in A \cdot P(x)$.

On dit que t est une ***preuve***, s'il existe une dérivation de $B \mid\!- t \in P$,où P est une proposition.

On remarque que l'on peut écrire $[x:A]$ seulement si A est un contexte ou une proposition.

Informellement la vérification de type est celle ci : (MN) est bien typé de type B si M a un type fonctionnel $[x:A]B$ et si N a un type A' egal à A à β-conversion près.

Il est important de noter que la consistence du système n'a été prouvée que pour ces trois niveaux de termes. X.LP Donnons tout de suite un exemple:

[A:*][x:A]* est un contexte.

[A:*][x:A]A est une proposition dont le type est le contexte précédent.On peut la lire comme $\forall A \cdot A \rightarrow A$.

[A:*][x:A]x est une preuve de la proposition précédente, c'est aussi l'application identité polymorphe.

1.2.3. Implantation

Nous décrivons ici le système que nous avons utilisé pour développer nos exemples.

L'implantation du système est réalisée en ML. Ce langage est à l'origine le méta-langage de LCF. C'est un lambda calcul typé, qui possède un opérateur de récursion. Une synthèse automatique des types est effectuée à chaque définition, en contrepartie le polymorphisme des types est plus restreint en ML que dans la théorie des constructions.

Trois types concrets représentent les constructions en ML: ***context*** (le type des contextes), ***object*** (le type des propositions et des preuves) et leur

somme *constr = context + object*.

Le langage des constructions permet de définir des constantes que l'on pourra utiliser par la suite. On peut donner un nom à toute expression bien typée. On déclare les constantes à l'aide de trois fonctions :

PROP : *string* $\rightarrow$ *object* $\rightarrow$ *void*
LET : *string* $\rightarrow$ *object* $\rightarrow$ *object* $\rightarrow$ *void*
LET_SYNTAX : *string* $\rightarrow$ *string list* $\rightarrow$ *void*

Si P est une proposition et t une preuve de P (en fait si P et t sont deux termes de Λ) alors :

PROP *name* P vérifie que P est une proposition valide, c'est-à- dire qu'il existe un contexte C tel que si B est l'environnement courant $B \mid- P \in C$ et lui affecte le nom *name* dans la théorie.
LET *name* t P vérifie que $B \mid- t \in P$ et donne le nom name à t.
Certains opérateurs courament utilisés admettent une syntaxe plus agréable.
LET_SYNTAX *name* $[l_1;l_2;...;l_n]$ permet d'écrire la construction qui a pour nom *name* appliquée aux arguments $x_1,x_2,...x_n,...x_{n+p}$ sous la forme

$$l_1 x_1 l_2 x_2 ... l_n x_n .. x_{n+p}$$

D'autre part on peut gérer l'environnement à l'aide de trois fonctions :

DECL *string* $\rightarrow$ *context* $\rightarrow$ *void*
AXIOM *string* $\rightarrow$ *object* $\rightarrow$ *void*
DISCHARGE *string* $\rightarrow$ *void*

DECL *name cont* vérifie que *cont* est un contexte dans l'environnement courant B et crée le nouvel environnement B,*name* :*cont*.
AXIOM *name propos* vérifie que *propos* est une proposition dans l'environnement courant B et crée le nouvel environnement B,*name* :*propos*
DISCHARGE *name* vérifie que l'environnent est de la forme B,*name* :L et transforme les constructions de M dans l'environnement B,*name* :L en des constructions de $[name:L]M$ dans l'environnement B.

On remarque que construire B dans un environnement $x_1:A_1,...,x_n:A_n$ revient à construire $[x_1:A_1]..[x_n:A_n]B$ dans un environnement vide. L'environnement apparait donc comme le cadre (hypothèses, axiomes, notations) de la preuve.

1.2.4. Abréviations

On utilisera les abréviations et notations suivantes:

!A,B.M	pour	[A:*][B:*]M
{P|C}	pour	[P:C] si C est un contexte
A$\rightarrow$B	pour	[x:A]B si x n'apparait pas dans B
(A B C)	pour	((A B) C)
let x=X in M_x	pour	M_Xi.e(([x:P] M_x) X) où X de type P

1.3. Introduction aux programmes

On trouvera en appendice les constructions de base, c'est-à- dire les connecteurs logiques, la définition des entiers, des booléens et des listes, ainsi qu'un certain nombre de preuves de résultats utiles dans la suite.

1.3.1. Les constructions de bases

La formation de ces structures de bases est assez intéressante car elle utilise l'analogie structure de donnée, structure de contrôle. Ainsi la conjonction de deux propositions A et B est définie comme :

$$!C.(A \rightarrow B \rightarrow C) \rightarrow C$$

Une preuve de $A\&B$ est donc un moyen de construire une preuve d'une proposition C quelconque dès que l'on sait construire une preuve de C à partir d'un élément de A et d'un élément de B.

A partir d'un élément h de $A\&B$ on construit un élément de type A (c'est la première projection). Pour cela il suffit d'appliquer h à A et à une preuve de $A \rightarrow B \rightarrow A$. La preuve de $A \rightarrow B \rightarrow A$ que nous utilisons est bien entendu $[x:A][y:B]x$. De même on construit la seconde projection, la formation de la paire ..etc..

Les programmes que nous présentons dans les prochains paragraphes sont écrits dans une théorie où se trouvent déjà les constructions de l'appendice. Les principales notations sont les suivantes:

- L'égalité (=) définie à la manière de Leibnitz :
  X.I !A.[x:A][y:A]{P|A->*}(P x)->(P y)
- La conjonction ($\&$) et les deux projections (*fst* et *snd*) ainsi que la formation de la paire ((,)).
- La somme disjointe (+) et les injections gauche et droite (*Inl* et *Inr*). L'axiome de construction de la somme disjointe est noté *axi_sum*, son type est :
  !A,B.[u:A+B]{P|A+B->}([a:A](P (Inl A B a)))->([b:B](P (Inr A B b)))->(P u)*
  Cet axiome dit qu'une preuve de $A+B$ est soit une injection à droite soit une injection à gauche. Il permet de prouver des propriétés sur les programmes construit à l'aide de sommes disjointes.
- Bottom, la proposition absurde est notée {} et est définie par *!A.A*, qui n'admet pas de preuve dans le système. La négation (~) est alors :
  !A.A->{}.
- La quantification existentielle (*Sig*) et son constructeur, *exist*, de type :
  !A.{P|A->}[a:A](P a)-><A>Sig(P)*
- Les entiers naturels (*Nat*), définis comme des itérateurs fonctionnels polymorphes :
  !A.(A->A)->A->A, on définit zero (0) et la fonction successeur (S). On utilise aussi l'axiome *peano* de type :
  [n:Nat]{P|Nat->}(P 0)->([u:Nat](P u)->(P (S u)))->(P n)*
  Il n'y a pas de preuve de cet axiome dans la théorie. On définit des relations d'ordre sur *Nat*. On note $n \leq m$ et $n \geq m$ respectivement ($LE\ n\ m$) et ($GE\ n\ m$). Posséder une preuve de ($LE\ n\ m$) ou de ($GE\ n\ m$) c'est avoir un moyen uniforme de prouver une propriété P pour m à partir d'une preuve de (Pn) et d'une preuve de stabilité de P de type : *[u:Nat](P u)->(P (S u))* pour *LE*, et *[u:Nat](P (S u))->(P u)* pour *GE*. On montre par ailleurs l'equivalence : *(LE n m)⇔(GE m n)* On introduit un autre axiome

noté *axi_ari* preuve de ~<*Nat*>(*S n*)=*0*.

- Les listes polymorphes, de type : *!A,B.(A->B->B)->B->B*, on dispose en particulier de la liste vide (*nil*) et de fonctions pour insérer un élément en tête de liste (*cons*, on note parfois *a.m* pour (*cons A a m*) dans les commentaires) et concaténer deux listes (*append*). Nous utilisons pour les listes un axiome de construction analogue à *peano* pour les entiers. Il est noté *axi_list*, son type est :
!A.[l:(list A)]{P|(list A)->}
(P (nil A))->([a:A][m:(list A)](P m)->(P (cons A a m)))->(P l)

1.3.2. Les principaux résultats de l'appendice

On rappellera la signification des preuves arithmétiques quand elles se présenteront. Ces preuves ne présentent pas beaucoup d'intérêt mais il était intéressant de trouver un noyau des principales propriétés des entiers et de l'addition permettant ensuite de prouver les résultats dont on avait besoin. Les preuves de base sur les listes sont l'associativité de *append* et la construction de deux fonctions *hd_tot* et *tl_tot* donnant respectivement l'élément de tête et le reste de la liste. Ces fonctions sont totales et ont donc des valeurs arbitraires pour la liste vide. Pour la somme disjointe on a montré en utilisant *axi_sum* que si u est de type $A+B$ avec $A \to B$ et si c_1 et c_2 sont deux éléments d'un type C alors soit $\varphi = (u\ C\ (A \to c_1)\ (B \to c_2))$ il existe deux preuves *cas_A* et *cas_B* de types respectifs $A \to$<*A*>*phhi*=*c* 1 et $B \to$<*A*>*phi*=*c* 2. Ainsi connait-on le comportement d'un programme construit par test sur deux possibilités qui s'excluent.

1.3.3. Les exemples

Dans les prochains paragraphes nous présentons quatre exemples de développement de programmes : des algorithmes de recherche de la fonction *lambo*(*f*) inverse d'une fonction *f* sur les entiers, l'algorithme MJRTY de Boyer et Moore [3] qui trouve un candidat majoritaire dans une liste de votes à l'aide d'un seul compteur, un programme de mise en page de texte (on sépare les mots d'une liste de caractères par un blanc ou un retour à la ligne de manière à mettre le maximum de mots par ligne) et enfin un programme de tri rapide (quicksort). Nous donnons l'écriture complète et commentée des programmes tels qu'ils sont vérifiés par le système.

2. La fonction lambo

2.1. Présentation

Cet exemple a été suggéré cette année dans un workshop sur les spécifications à Göteborg. Il fallait écrire la spécification d'un programme puis le développer. La tâche à effectuer est la suivante :
Soit f une fonction de N dans N, croissante et non bornée, trouver la fonction $lambo(f)$ définie par :

$$lambo\ (f)\ (x) = inf\ \{\ y \in N \mid f(y) > x\ \}$$

la proposition à prouver, et par conséquent notre spécification est :

$$\forall x \in N,\ \exists y \in N\ tel\ que\quad f(y) > x\ et\ \forall z \in N \cdot f(z) > x \Rightarrow y \leq z$$

Une preuve mathématique d'un tel raisonnement est :
$\forall x \in N\ \{\ y \in N \mid f(y) > x\ \}$ est une partie de N non vide car f est non bornée (on notera dans la suite z_x un de ses éléments), elle admet donc un plus petit élément (noté inf_x).

D'un point de vue informatique on s'intéresse à la construction de cet élément. Il y a pour cela plusieurs méthodes qui correspondent à différents algorithmes donc à différentes preuves du résultats. Nous avons développé ici quatre programmes dont nous donnons sommairement les principes.

1 $inf_x \in [0..z_x]$ et inf_x est le premier élément en partant de 0 qui vérifie $f(z) > x$. Ce procédé de recherche s'applique à un prédicat quelconque P sur N pourvu qu'il soit décidable et que $\exists y \in N \cdot P(y)$.

2-3 $\{\ y \in N \mid f(y) > x\ \}$ est un segment car f est croissante inf_x est donc caractérisé par $f(inf_x) > x$ *et* $f(inf_x - 1) \leq x$. On peut le trouver de deux manières soit par une recherche décroissante à partir de z_x (méthode 2), soit par une recherche dichotomique (méthode 3). Remarquons que la méthode 2 pourrait aussi s'appliquer à un prédicat plus général P, à condition de continuer la recherche jusqu'à 0. Cet algorithme n'est pas développé ici puisqu'il nécessite au moins autant de tests que le premier.

4 Cette méthode est suggérée par la forme de la spécification. On veut construire un élément en fonction d'un entier x. On peut essayer de le construire par récurrence sur x. Ici le seul lien entre inf_x et inf_{x+1} est $inf_x \leq inf_{x+1}$. On peut donc raffiner une des méthodes 1 ou 3 en cherchant inf_{x+1} entre inf_x (au lieu de 0) et z_x. Nous présentons ici la méthode 1 améliorée, l'algorithme est intéressant seulement dans le cas où l'on cherche à calculer successivement les valeurs de $lambo(f)$ en effet sinon l'appel récursif redonne exactement la méthode 1.

Tous ces programmes se construisent en fait par itération, on prouve donc par diverses structures de récurrence une certaine propriété (différente pour chaque méthode) dont on déduit la spécification générale du programme. Cet exemple est très simple mais donne une bonne illustration de

l'utilisation du calcul des constructions en tant que langage de programmation de très haut niveau.

2.2. Programme

Nous donnons maintenant le programme, tel qu'il a été écrit pour la machine. Les commentaires sont en italiques.

DEFINITION DE PROPRIETES.

Fonction croissante :

```
PROP 'increas' "[f:Nat->Nat][n:Nat][m:Nat](LT n m)->(LE (f n) (f m))";;
```

Fonction non bornée :

```
let UNBOUND = "[f:Nat->Nat][x:Nat] <Nat>Sig([y:Nat] (LT x (f y)))";;
PROP 'unbound' UNBOUND;;
```

Définition de la proposition :
Soit P un prédicat sur Nat et n un entier, tous les entiers vérifiant P sont plus grands que n :

```
let OVER = "{P|Nat->*}[n:Nat][z:Nat](P z)->(LE n z)";;
PROP 'over' OVER;;
```

Définition de la proposition :
Soit P un prédicat sur Nat, y est le plus petit entier vérifiant P :

```
let SMALL = "{P|Nat->*}[y:Nat](P y)&(over P y)";;
PROP 'small' SMALL;;
```

AXIOMES

Décidabilité de la relation ≤.

```
AXIOM 'LE_dec' "[n:Nat][m:Nat](LE n m)+(~(LE n m))";;
```

Le reste du programme est développé dans un environnement où une fonction f sur les entiers est donnée.

```
AXIOM 'f' "Nat->Nat";;
```

SPECIFICATION DU PROGRAMME

Dans notre cas, x étant fixé le prédicat est y → x<f (y).

```
PROP 'P_lamb' "[x:Nat][y:Nat](LT x (f y))";;
```

Définition du prédicat lambo, (lambo x y) signifie que y est le plus petit entier d'image par f supérieure à x.

```
let LAMBO = "[x:Nat](small (P_lamb x))";;
PROP 'lambo' LAMBO;;
```

Spécification :

```
let LAMBSPEC = "(increas f)->(unbound f)->[x:Nat]<Nat>Sig((lambo x))";;
PROP 'lamb_spec' LAMBSPEC;;
```

RESULTATS PRELIMINAIRES :

Tous les entiers étant supérieurs à 0, on a une preuve de (over P 0) :

```
let OVER0 = "{P|Nat->*}[z:Nat][h:(P z)](peano z)";;
LET 'over0' OVER0 "{P|Nat->*}(over P 0)";;
```

La décidabilité de ≤ assure celle de P_lamb. (En fait on a juste besoin de la décidabilité de P_lamb et non pas de celle plus générale de ≤).

```
let P_LAMB_DEC = "[x:Nat][y:Nat](LE_dec (S x)(f y))";;
LET 'P_lamb_dec' P_LAMB_DEC "[x:Nat][y:Nat]
                    (P_lamb x y)+(~(P_lamb x y))";;
```

Résultats sur les puissances de 2.

Définition de la fonction $n \to 2^n$, c'est l'itération n fois de la multiplication par 2 à partir de 1 :

```
let POW2 = "[n:Nat](n Nat mult2 1)";;
LET 'pow2' POW2 "Nat->Nat";;
```

Fonction qui à u et n associe $u+2^n$:

```
LET 'add_pow' "[u:Nat][n:Nat](add u (pow2 n))" "Nat->Nat->Nat";;
```

Preuve de $0<2^n$:
LE_n_n est une preuve de $\forall n \in N.\ n \leq n$
stab_add_r est une preuve de : $\forall a,b,c \in N\ a \leq b \Rightarrow a+c \leq b+c$.

```
let LT_0_POW2 = "[n:Nat](peano n ([v:Nat](LT 0 (pow2 v)))
            (LE_n_n 1)
            ([v:Nat][h:(LT 0 (pow2 v))]
            (trans_LE (S 0) (pow2 v) (pow2 (S v)) h
            (stab_add_r 0 (pow2 v) (pow2 v) (peano (pow2 v))))))";;
LET 'LT_0_pow2' LT_0_POW2 "[n:Nat](LT 0 (pow2 n))";;
```

Preuve de $n \to 2^n$ est une fonction strictement croissante :

```
let POW2_CREAS = "[n:Nat]
            (stab_add_r (S 0) (pow2 n) (pow2 n) (LT_0_pow2 n))";;
LET 'pow2_creas' POW2_CREAS "[n:Nat](LT (pow2 n) (pow2 (S n)))";;
```

Cette dernière propriété permet de montrer que tout entier est majorable par une puissance de deux. Cette preuve est aussi un algorithme pour trouver le majorant. On pourrait utiliser une preuve de $n<2^n$, ici on obtient en fait la plus petite puissance de 2 majorante.
LE_EGLT est une preuve de $\forall n,m \in N.n \leq m \Rightarrow n=m$ ou $n<m$

```
PROP 'MAJ_PROP' "[x:Nat][n:Nat](LE x (pow2 n))";;
```

```
let POW2_MAJ = "[y:Nat](peano y ([x:Nat]<Nat>Sig((MAJ_PROP x)))
        (exist Nat (MAJ_PROP 0) 0 (LE_n_Sn 0))
        ([x:Nat][hyp:<Nat>Sig((MAJ_PROP x))]
        (hyp <Nat>Sig((MAJ_PROP (S x)))
        ([n:Nat][h:(LE x (pow2 n))]
        (LE_EGLT x (pow2 n) h <Nat>Sig((MAJ_PROP (S x)))
        ([t1:<Nat>x=(pow2 n)]
        (exist Nat (MAJ_PROP (S x)) (S n)
        (sym Nat x (pow2 n) t1 ([z:Nat](MAJ_PROP (S z) (S n)))
                (pow2_creas n))))
        ([t2:(LT x (pow2 n))]
        (exist Nat (MAJ_PROP (S x)) n t2)))))))";;

LET 'pow2_maj' POW2_MAJ "[y:Nat]<Nat>Sig((MAJ_PROP y))";;
```

Propriétes liées à la croissance de f :

Croissance large : $x \leq y \Rightarrow f(x) \leq f(y)$.

```
let INCR_LE = "[f1:(increas f)][x:Nat][y:Nat][h:(LE x y)]
        (LE_EGLT x y h (LE (f x) (f y))
        ([t1:<Nat>x=y]
        (t1 ([z:Nat](LE (f x) (f z))) (LE_n_n (f x))))
        ([t2:(LT x y)] (f1 x y t2)))";;
LET 'incr_LE' INCR_LE "(increas f)->[x:Nat][y:Nat](LE x y)->(LE (f x)(f y))";;
```

Si $f(y) \leq x$ alors $\forall z \cdot z \leq y \Rightarrow f(z) \leq x$.

```
let CREAS_INF = "[hyp:(increas f)][x:Nat][y:Nat]
        [h1:(LE (f y) x)][z:Nat][h2:(LE z y)]
        (trans_LE (f z) (f y) x (incr_LE hyp z y h2) h1)";;
let CRE_PROP = "(increas f)->[x:Nat][y:Nat]
        (LE (f y) x)->[z:Nat](LE z y)->(LE (f z) x)";;
LET 'creas_inf' CREAS_INF  CRE_PROP;;
```

Si $x \leq f(y)$ alors $\forall z \cdot y \leq z \Rightarrow x \leq f(z)$:

```
let CREAS_SUP = "[hyp:(increas f)][x:Nat][y:Nat][h1:(LE x (f y))][z:Nat]
        [h2:(LE y z)]
        (trans_LE x (f y) (f z) h1 (incr_LE hyp y z h2))";;
LET 'creas_sup' CREAS_SUP "(increas f)->[x:Nat][y:Nat]
                (LE x (f y))->[z:Nat](LE y z)->(LE x (f z))";;
```

PREMIERE RESOLUTION

Le principe est x étant fixé, tester tous les entiers à partir de 0. Le premier z vérifiant f(z)>x convient. Cette méthode ne fait pas intervenir la croissance de f, on va la prouver de manière générale : soit P un prédicat décidable sur Nat, on suppose qu'il existe y tel que P(y), trouver le plus petit y vérifiant P. On commence par montrer par récurrence sur n que : $\forall n \in N$, soit on a construit le plus petit entier vérifiant P, soit tous les entiers vérifiant P sont supérieurs à n.

Environnement : un prédicat P et la preuve de sa décidabilité.

```
DECL 'P' "Nat->*";;
AXIOM 'P_dec' "[y:Nat](P y)+(~(P y))";;
```

Spécificaton intermédiaire :

```
let INTER = "[y:Nat]<Nat>Sig((small P))+(over P y)";;
PROP 'interm1' INTER;;
```

On définit deux abréviations pour les injections à gauche et à droite dans la somme disjointe de cette spécification.

```
let cas1 = "[y:Nat](Inl <Nat>Sig((small P)) (over P y))";;
LET 'cas1' cas1 "[y:Nat]<Nat>Sig((small P))->(interm1  y)";;

let cas2 = "[y:Nat](Inr <Nat>Sig((small P)) (over P y))";;
LET 'cas2' cas2 "[y:Nat](over P y)->(interm1  y)";;
```

Preuve du cas de base de la récurrence :

```
let INTER_0 = "(cas2 0 (over0 P))";;
LET 'inter_0' INTER_0 "(interm1 0)";;
```

Preuve du pas de récurrence :

```
let INTER_REC = "[n:Nat][hyp_rec:(interm1 n)]
            (hyp_rec (interm1 (S n))
            ([x1:<Nat>Sig((small P))](cas1 (S n) x1))
            ([x2:(over P n)]
            (P_dec n (interm1 (S n))
            ([t1:(P n)] (cas1 (S n) (exist Nat (small P) n
               <(P n),(over P n)>(t1,x2))))
            ([t2:~(P n)](cas2 (S n)
            ([z:Nat][h:(P z)](LE_EGLT n z (x2 z h) (LE (S n) z)
                        ([t21:<Nat>n=z]
                        (t21 ([u:Nat]~(P u)) t2 h (LE (S n) z)))
                        ([t22:(LT n z)] t22))))))))";;
LET 'inter_rec' INTER_REC "[n:Nat](interm1 n)->(interm1 (S n))";;

let INTER_PROOF = "[n:Nat](peano n (interm1)
              (inter_0 ) (inter_rec ))";;
LET 'prop_inter' INTER_PROOF "interm1";;
```

D'un point de vue programmation la preuve précédente correspond au programme suivant :

```
var fin : bool; n : integer;
n:=0; fin:=false;  (cas de base)
  while not(fin) do if P(n) then fin:=true
                    else n:=n+1.
```

L'hypothèse <Nat>Sig(P) assure la terminaison du programme.

```
let TERMIN = "[hyp:<Nat>Sig(P)]
          (hyp <Nat>Sig((small P))
          ([y:Nat][h1:(P y)]
          (prop_inter  y <Nat>Sig((small P))
          <<Nat>Sig((small P))>Id
          ([h2:(over P y)]
             (exist Nat (small P) y
             <(P y),(over P y)>(h1,h2))))))";;
LET 'smallP_proof' TERMIN "<Nat>Sig(P)-><Nat>Sig((small P))";;
```

Il nous reste à appliquer ceci à notre problème ,on décharge les hypothèses P_dec et P . On construit alors le terme :

```
"[hyp1:(increas f)][hyp2:(unbound f)][x:Nat]
 (smallP_proof (P_lamb x) (P_lamb_dec x) (hyp2 x))"
```

le type de ce terme est lamb_spec.

DEUXIEME RESOLUTION

Le principe de cette méthode est de commencer par construire un élement y tel que $f(y)>x$. On appelle $z(x)$ un tel élément. La suite de la preuve correspond à un programme :

var z:=integer;
z:=z(x);
while z>0 and x<f(z-1) do z:=z-1;

x étant fixé, on montre par une récurrence descendante sur n que la propriété: pour tout n, soit $x<f(n)$, soit on a trouvé le plus entier vérifiant le prédicat *P_lamb*, est vraie pour 0.

Spécification intermédiaire :

```
let INTER = "[x:Nat][n:Nat]((LT x (f n))+<Nat>Sig((lambo x)))";;
PROP 'interm2' INTER;;
```

Les deux injections liées à cette spécification :

```
let cas1 = "[x:Nat][n:Nat](Inl (LT x (f n)) <Nat>Sig((lambo x)))";;
LET 'cas1' cas1 "[x:Nat][n:Nat](LT x (f n))->(interm2 x n)";;

let cas2 = "[x:Nat][n:Nat](Inr (LT x (f n)) <Nat>Sig((lambo x)))";;
LET 'cas2' cas2 "[x:Nat][n:Nat] <Nat>Sig((lambo x))->(interm2 x n)";;
```

Pas de la récurrence descendante :(interm2 x S(n))->(interm2 x n).

```
let INTER_REC = "[hyp:(increas f)][x:Nat][n:Nat][hyp_rec:(interm2 x (S n))]
   (hyp_rec (interm2 x n)
   ([t1:(LT x (f (S n)))](P_lamb_dec x n (interm2 x n)
         ([t11:(LT x (f n))](cas1 x n t11))
         ([t12:~(LT x (f n))]
   let overSn = [z:Nat][h:(LT x (f z))](LE_ou_LT z n (LE (S n) z)
         ([p:(LE z n)](t12 (creas_sup hyp (S x) z h n p) (LE (S n) z)))
                              <(LT n z)>Id) in
         (cas2 x n (exist Nat (lambo x) (S n)
         <(LT x (f (S n))),(over (P_lamb x) (S n))>(t1,overSn))))))
   ([t2:<Nat>Sig((lambo x))](cas2 x n t2)))";;

LET 'inter_rec' INTER_REC
      "(increas f)->[x:Nat][n:Nat](interm2 x (S n))->(interm2 x n)";;
```

L'hypothèse f non bornée fournit le cas de base, une récurrence descendante prouve que (interm2 x) est vrai en 0. Il reste à montrer que l'on a alors résolu le problème.

```
let PROG_0 = "[hyp1:(increas f)][hyp2:(unbound f)][x:Nat]
          (hyp2 x (interm2 x 0) ([y:Nat][h:(LT x (f y))]
          (rec_inv y (interm2 x) (cas1 x y h)
          (inter_rec hyp1 x))))";;
LET 'prog_0' PROG_0
      "(increas f)->(unbound f)->[x:Nat](interm2 x 0)";;
```

```
let PROGRAM = "[hyp1:(increas f)][hyp2:(unbound f)][x:Nat]
        (prog_0 hyp1 hyp2 x <Nat>Sig((lambo x))
        ([x1:(LT x (f 0))](exist Nat (lambo x) 0
        <(LT x (f 0)),(over (P_lamb x) 0)>(x1,(over0 (P_lamb x)))))
        <<Nat>Sig((lambo x))>Id)";;
LET 'prog_decroissant' PROGRAM "lamb_spec";;
```

TROISIEME RESOLUTION

On prouve une méthode de recherche dichotomique. La spécification intermédiaire utilisée est que soit $x<f(0)$, soit on peut raisonner par récurrence descendante sur la proposition suivante :

$$\forall n \in N \; \exists u \in N \text{ tel que } x \geq f(u) \text{ et } x < f(u+2^n) .$$

Une abréviation pour la propriété $x \geq f(u)$ et $x < f(u+2^n)$.

```
let ENCADR = "[x:Nat][n:Nat][u:Nat]
        ~(LT x (f u))&(LT x (f (add_pow u n)))";;
PROP 'encadr' ENCADR ;;
```

On introduit les deux projections correspondantes ainsi que la formation de la paire.

```
let PRO1= "[x:Nat][n:Nat][u:Nat][h:(encadr x n u)]
      <~(LT x (f u)),(LT x (f (add_pow u n)))>fst(h)";;
LET 'pro1' PRO1 "[x:Nat][n:Nat][u:Nat](encadr x n u)->(~(LT x (f u)))";;

let PRO2 = "[x:Nat][n:Nat][u:Nat][h:(encadr x n u)]
      <~(LT x (f u)),(LT x (f (add_pow u n)))>snd(h)";;
LET 'pro2' PRO2 "[x:Nat][n:Nat][u:Nat]
        (encadr x n u)->(LT x (f (add_pow u n)))";;

let PAIR = "[x:Nat][n:Nat][u:Nat]
       [h1:~(LT x (f u))][h2:(LT x (f (add_pow u n)))]
       <~(LT x (f u)),(LT x (f (add_pow u n)))>(h1,h2)";;
LET 'pair' PAIR "[x:Nat][n:Nat][u:Nat]
        ~(LT x (f u))->(LT x (f (add_pow u n)))->(encadr x n u)";;
```

Spécification intermédiaire :

```
let DICHO = "[x:Nat][n:Nat]<Nat>Sig((encadr x n))";;
PROP 'dicho' DICHO;;
```

Si $f(0)>x$ on n'a jamais (encadr x n) mais on a directement le résultat sinon on trouve un premier encadrement en utilisant f non bornée, la croissance de f, et la preuve pow2_maj que tout nombre peut être majoré par une puissance de 2 .

Le cas $f(0)>x$:

```
let CAS_LT_0 = "[x:Nat][h:(LT x (f 0))]
        (exist Nat (lambo x) 0
        <(LT x (f 0)),(over (P_lamb x) 0)>(h,(over0 (P_lamb x))))";;
LET 'cas_LT_0' CAS_LT_0 "[x:Nat](LT x (f 0))-><Nat>Sig((lambo x))";;
```

Le cas $\sim(x<f(0))$: on obtient alors un premier encadrement.

```
let CAS_noLT_0 = "[f1:(increas f)][f2:(unbound f)][x:Nat][h1:~(LT x (f 0))]
                  (f2 x <Nat>Sig((dicho x))
                  ([y:Nat][h2:(LT x (f y))]
                  (pow2_maj y <Nat>Sig((dicho x))
                  ([n:Nat][h3:(LE y (pow2 n))]
                  (exist Nat (dicho x) n
                  (exist Nat (encadr x n) 0
                  (pair x n 0 h1
                  (creas_sup f1 (S x) y h2 (pow2 n) h3))))))))";;
LET 'cas_noLT_0' CAS_noLT_0
  "(increas f)->(unbound f)->[x:Nat][h1:~(LT x (f 0))]<Nat>Sig((dicho x))";;
```

Pas de récurrence :

```
let DICHO_REC = "[x:Nat][n:Nat][h1:(dicho x (S n))]
                 (h1 (dicho x n)
                 ([u:Nat][h2:(encadr x (S n) u)]
                 (P_lamb_dec x (add_pow u n) (dicho x n)
                 ([t1:(LT x (f (add_pow u n)))]
                 (exist Nat (encadr x n) u
                 (pair x n u (pro1 x (S n) u h2) t1)))
                 ([t2:~(LT x (f (add_pow u n)))]
                 (exist Nat (encadr x n) (add_pow u n)
                 (pair x n (add_pow u n) t2
                 (add_assoc_r u (pow2 n) (pow2 n)
                 ([z:Nat](LT x (f z))) (pro2 x (S n) u h2))))))))";;
LET 'dicho_rec' DICHO_REC "[x:Nat][n:Nat](dicho x (S n))->(dicho x n)";;
```

Prouver (dicho x 0) c'est trouver u tel que f(u)≤x et x<f(S(u)) La croissance de f permet d'en déduire que S(u) est l'élément cherché.

```
let DICHO_LAMBO = "[f1:(increas f)][x:Nat][h:(dicho x 0)]
                   (h <Nat>Sig((lambo x))
                   ([u:Nat][hyp:(encadr x 0 u)]
    let cond1 = (sym_add u 1 ([z:Nat](LT x (f z))) (pro2 x 0 u hyp)) in
    let cond2 = [z:Nat][p:(LT x (f z))]
                   (LE_ou_LT z u (LE (S u) z) ([t:(LE z u)]
                   (pro1 x 0 u hyp (creas_sup f1 (S x) z p u t) (LE (S u) z)))
                                  <(LT u z)>Id) in
                   (exist Nat (lambo x) (S u)
    <(LT x (f (S u))),(over (P_lamb x)(S u))>(cond1,cond2))))";;
LET 'dicho_lambo' DICHO_LAMBO "(increas f)->[x:Nat]
                               (dicho x 0)-><Nat>Sig((lambo x))";;
```

Synthèse de la preuve :

```
let DICHO_PROOF = "[h1:(increas f)][h2:(unbound f)][x:Nat]
                   (P_lamb_dec x 0 <Nat>Sig((lambo x))
                   (cas_LT_0 x)
                   ([t:~(LT x (f 0))](cas_noLT_0 h1 h2 x t <Nat>Sig((lambo x))
                   ([n:Nat][h3:(dicho x n)]
                   (dicho_lambo h1 x
                   (rec_inv n (dicho x) h3 (dicho_rec x)))))))";;
LET 'dicho_proof' DICHO_PROOF "lamb_spec";;
```

QUATRIEME RESOLUTION

On calcule *lambo*$(f)(x)$ par récurrence sur x, en utilisant la propriété: $x+1<f(y) \Rightarrow x<f(y)$, le même programme s'applique à un prédicat $P(x,y)$ tel que $P(x+1,y) \Rightarrow P(x,y)$. Le principe est celui de la première résolution. Ce programme est meilleur si on cherche à calculer successivement les valeurs de *lambo*(f).

La spécification intermédiaire est analogue à celle de la première résolution.

```
let INTER = "[x:Nat][y:Nat]<Nat>Sig((lambo x))+(over (P_lamb x) y)";;
PROP 'interm4' INTER;;
```

On définit les injections correspondantes.

```
let cas1 = "[x:Nat][y:Nat](Inl <Nat>Sig((lambo x))(over (P_lamb x) y))";;
LET 'cas1' cas1 "[x:Nat][y:Nat]
        <Nat>Sig((lambo x))-><Nat>Sig((lambo x))+(over (P_lamb x) y)";;

let cas2 = "[x:Nat][y:Nat](Inr <Nat>Sig((lambo x))(over (P_lamb x) y))";;
LET 'cas2' cas2 "[x:Nat][y:Nat]
          (over (P_lamb x) y)-><Nat>Sig((lambo x))+(over (P_lamb x) y)";;
```

On utilise plusieurs fois le résultat suivant :
si (over (P_lamb x) n) et (P_lamb x n) alors <Nat>Sig((lambo x)).

```
let END = "[x:Nat][n:Nat][h1:(over (P_lamb x) n)][h2:(P_lamb x n)]
      (exist Nat (lambo x) n <(P_lamb x n),(over (P_lamb x) n)>(h2,h1))";;
LET 'end' END "[x:Nat][n:Nat]
          (over (P_lamb x) n)->(P_lamb x n)-><Nat>Sig((lambo x))";;
```

Preuve de S(x)<f(y) ⇒ x<f(y) :

```
let LAMB_S = "[x:Nat][z:Nat][h:(LT (S x) (f z))]
        (trans_LE (S x) (S (S x)) (f z) (LE_n_Sn (S x)) h)";;
LET 'lamb_S' LAMB_S "[x:Nat][z:Nat](P_lamb (S x) z)->(P_lamb x z)";;
```

Le passage de (interm4 x n) à (interm4 x (S n)) se fait comme dans le premier programme en testant (P_lamb x n).

```
let INTER_HER = "[x:Nat][n:Nat][hyp_rec:(interm4 x n)]
          (hyp_rec (interm4 x (S n))
          ([x1:<Nat>Sig((lambo x))](cas1 x (S n) x1))
          ([x2:(over (P_lamb x) n)]
          (P_lamb_dec x n (interm4 x (S n))
          ([t1:(P_lamb x n)] (cas1 x (S n) (end x n x2 t1)))
          ([t2:~(P_lamb x n)](cas2 x (S n)
          ([z:Nat][h:(P_lamb x z)](LE_EGLT n z (x2 z h) (LE (S n) z)
                    ([t21:<Nat>n=z]
                (t21 ([u:Nat]~(P_lamb x u)) t2 h (LE (S n) z)))
                    ([t22:(LT n z)] t22))))))))";;
LET 'inter_her' INTER_HER "[x:Nat](her (interm4 x))";;
```

Recherche de lambo(f)(0), c'est exactement le premier programme dans ce cas particulier :

```
let LAMBO_0 = "[hyp:(unbound f)]
        let base = (cas2 0 0 (over0 (P_lamb 0))) in
        (hyp 0 <Nat>Sig((lambo 0))
        ([y:Nat][h1:(P_lamb 0 y)]
        (peano y (interm4 0) base (inter_her 0) <Nat>Sig((lambo 0))
            <<Nat>Sig((lambo 0))>Id
            [h2:(over (P_lamb 0) y)](end 0 y h2 h1))))";;
LET 'lambo_0' LAMBO_0 "(unbound f)-><Nat>Sig((lambo 0))";;
```

L'utilisation de l'hypothèse de récurrence permet de montrer (interm4 (S x)) en testant les entiers à partir de lambo(f)(x), la terminaison de ce procédé est assurée comme précédemment par unbound (f). On commence par montrer (interm4 (S x) (lambo(f)(x))).

```
let INTER_LAMB = "[x:Nat][inf:Nat][h1:(over (P_lamb x) inf)]
        (cas2 (S x) inf
        ([z:Nat][h2:(LT (S x) (f z))](h1 z (lamb_S x z h2))))";;
LET 'inter_lamb' INTER_LAMB "[x:Nat][inf:Nat]
                    (over (P_lamb x) inf)->(interm4 (S x) inf)";;

let LAMBO_REC = "[hyp:(unbound f)][x:Nat][hyp_rec:<Nat>Sig((lambo x))]
        (hyp_rec <Nat>Sig((lambo (S x)))
        ([inf:Nat][h1:(lambo x inf)]
        (h1 <Nat>Sig((lambo(S x)))
        ([h11:(P_lamb x inf)][h12:(over (P_lamb x) inf)]
        (hyp (S x) <Nat>Sig((lambo (S x)))
        ([y:Nat][h2:(LT (S x) (f y))](h12 y (lamb_S x y h2)
        (interm4 (S x))
        (inter_lamb x inf h12) (inter_her (S x))
        <Nat>Sig((lambo (S x)))
        <<Nat>Sig((lambo (S x)))>Id
        ([h3:(over (P_lamb (S x)) y)](end (S x) y h3 h2)))))))))";;
LET 'lambo_rec' LAMBO_REC "[hyp:(unbound f)][x:Nat]
                    <Nat>Sig((lambo x))-><Nat>Sig((lambo(S x)))";;
```

Synthèse de la preuve. Elle correspond au programme suivant :

```
function lambo(x:integer):integer
var y:integer; fin:bool;
begin
fin := false;
if x=0 then begin y:=0;
        while not fin do
          if 0<f(y) then begin fin:=true; lambo(0):=y end
                else y:=y+1
      end
   else begin y:=lambo(x-1);
        while not fin do
          if x<f(y) then begin fin:=true; lambo(x):=y end
                else y:=y+1
        end
   end;
```

```
let PROGRAM = "[hyp1:(increas f)][hyp2:(unbound f)][x:Nat]
        (peano x ([y:Nat]<Nat>Sig((lambo y)))
        (lambo_0 hyp2) (lambo_rec hyp2))";;
LET 'prog_rec' PROGRAM "lamb_spec";;
```

2.3. Remarques

Cet exemple, très simple, nous permet déjà de faire un certain nombre de remarques sur le calcul des constructions. Remarquons tout d'abord la souplesse d'expression. Il n'y a aucune difficulté à définir de nouveaux concepts tels que les fonctions non bornées, le plus petit élément vérifiant un prédicat etc. Par contre ce calcul souffre d'un certaine lourdeur syntaxique due en particulier aux opérateurs polymorphes comme la paire et les projections. Dans [6], Th. Coquand et G. Huet expliquent comment on peut synthétiser automatiquement certains de ces arguments. Voici un exemple d'une preuve extrèmement simple mais dont l'écriture est bien longue:
Soit H de type : $(<A>a=b)\&(<A>a=c)$. Une preuve de la proposition $<A>b=c$ est:

(trans A b a c
(sym A a b <<A>a=b,<A>a=c>fst(H)) <<A>a=b,<A>a=c>snd(H))

avec *trans* et *sym* des preuves de la transitivité et de la symétrie de l'égalité. Evidemment, cette lourdeur cache parfois au lecteur non habitué la simplicité des idées exprimées et le caractère très naturel des démonstrations. Mais cela ne doit pas être un obstacle. En effet, on peut envisager de génerer ce type de construction à l'aide de formules simples, soit par des méthodes telles que la synthèse automatique d'arguments du polymorphismes, soit par une synthèse de programmes.

Remarquons également les liens entre preuve et programme. En écrivant un programme comme une preuve, on utilise le calcul des constructions comme un langage de très haut niveau. En effet, on a une vérification mécanique de la spécification des différentes parties de notre programme. Ce qui dans d'autres langages apparait en commentaires est ici pris en compte. On distingue différents niveaux de programmes dans le λ-terme que l'on écrit. Tout d'abord le λ-terme pur que l'on peut extraire en supprimant les types et la partie logique (c'est-à-dire les propositions, cf [5] pour une définition formelle), on exécute le programme en réduisant ce λ-terme. Ceci est le niveau le plus bas du programme. On peut aussi utiliser un niveau intermédiaire avec une structure de type sans produit dépendant. Ainsi on peut écrire une fonction qui calcule le plus petit entier vérifiant un prédicat P. Il faut toujours supposer que P est décidable soit *P_dec* une preuve de *[n:Nat](P n)+~(P n)* et soit z un entier que l'on suppose "moralement" vérifier P alors:
trouve = (z Nat ([n:Nat](P_dec n Nat ((P n)->n) (~(P n)->(S n))) 0))
est un programme qui fournit un entier qui "moralement" est le plus petit vérifiant P. Faisons deux remarques. Tout d'abord on voit apparaitre un entier z qui donne le nombre d'itérations de la fonction. Dans un langage de programmation plus classique serait apparue une structure de contrôle de type *while*, mais tous les programmes du calcul des constructions terminent du fait du théorème de normalisation. D'où la nécessité de borner à priori le nombre d'itérations. La deuxième remarque est l'analogie entre ce petit programme d'une ligne et celui que nous donnons pour la première résolution du problème. Le programme prouvé a la même structure que la fonction *trouve* mais a été enrichi par l'aspect preuve. Ainsi on n'utilise pas un entier z mais une preuve de $<Nat>Sig(P)$ c'est-à-dire en fait z et une preuve de (Pz). On n'itère pas sur les entiers mais sur une propriété des entiers à savoir:
[n:Nat]<Nat>Sig((P))+([p:Nat](P p)->(LE n p) Prouver une propriété par itération, c'est exactement faire une preuve par récurrence, d'où l'utilisation de *(peano z)*. Evidemment ceci a multiplié le nombre de lignes de notre

programme mais on dispose d'une preuve complète du programme qui est vérifiée mécaniquement, qui est plus aisée à écrire qu'une preuve à posteriori car on se laisse guider par les types et qui enfin permet d'améliorer facilement le programme puisque les raisons de son fonctionnement sont apparentes. D'où des possibilités d'abstraction et de simplification des hypothèses. Par exemple, dans la première résolution l'hypothèse f croissante n'apparait pas, l'abstraction par rapport à un prédicat unaire allège l'écriture du programme, l'hypothèse f non bornée donne la preuve de terminaison tandis qu'elle est utilisée comme procédure effective dans les deuxième et troisième résolutions.

Il est cependant important de pouvoir écrire dans le calcul des constructions des programmes simples comme par exemple l'addition ou les puissances de 2. Sur ces programmes on prouve à posteriori certaines propriétés soit par β-réduction, c'est-à-dire en calculant, soit en regardant plus en profondeur dans la structure du terme, par exemple en raisonnant par récurrence dans le cas de programmes écrits par application d'un entier.

Cet exemple nous permet de regarder comment se traduisent les structures de contrôle itératives des langages de programmation classiques dans le calcul des constructions. Les boucles PASCAL *for i:=n to m do .., for j:=n downto m do...* se traduisent respectivement par l'application des preuves de $(LE\ n\ m)$ et $(GE\ n\ m)$ à un certain prédicat P sur les entiers à une preuve de $(P\ n)$ et à une preuve de stabilité croissante ou décroissante de P. On obtient ainsi une preuve de $(P\ m)$. La proposition $(P\ m)$ peut être directement la spécification du programme qui est en général de la forme $<A>Sig(Q)$. La preuve de (Pn) nous donne un a_n qui convient au départ (initialisation), la stabilité de P fournit le passage de a_p à a_{p-1} ou a_{p+1} plus les preuves des $Q(a_i)$. On peut aussi avoir P de la forme $[n:Nat]<A>Sig(Q)+(R\ n)$ avec $<A>Sig(Q)$ la spécification du programme. Ceci correspond à une boucle "while" dont $(R\ n)$ serait l'invariant. Prouver la terminaison du programme c'est prouver que $<A>Sig(Q)+(R\ m)->$

On voit ainsi apparaitre une possibilité de systématiser certains schémas de preuves de programmes.

3. Algorithme de vote

3.1. Présentation

Nous présentons le développement d'un algorithme de vote, MJRTY, dû à Boyer et Moore [3]. Nous nous sommes inspirés d'un travail similaire réalisé par R.Backhouse dans le système de Martin-Löf [1]. Si la démarche générale est la même, notre développement est lui, totalement mécanisé.

Cet algorithme détermine l'existence d'un candidat majoritaire dans une liste de votes en ne comptant les voix que d'un seul candidat. Ceci peut être intéressant si on ne connait pas à priori le nombre de candidats et donc le nombre de compteurs à utiliser. Le principe de l'algorithme est simple : on fait entrer les votants tour à tour dans une pièce, s'il y a deux personnes d'opinions contraire alors ils s'éliminent. S'il existe un candidat majoritaire alors il restera l'un de ses partisans à la fin de la bagarre. Cela revient en fait à considérer qu'à chaque instant il n'y a que 2 candidats, celui qui a peut-être la majorité et un autre et à se dire que toute situation est équivalente d'un point de vue des majorités possible à la situation où la moitié des gens à terre sont pour le candidat majoritaire possible, les autres étant pour le "second" candidat imaginaire. Formellement, on dispose d'un tableau $a[1..n]$ où sont inscrits les votes. Une variable x contient la voix du vainqueur du moment. Elle est initialisée par un candidat x_0 quelconque, e est une variable entière initialisée à 0. On effectue :

```
for i:=1 to n do
    if x=a[i] then e:=e+1
          else if i=2e then begin
                            x:=a[i];
                            e:=e+1
                            end;
```

Ce programme, à priori peu simple à prouver, apparait assez naturellement après plusieurs réductions de la spécification générale du problème. Ce programme apparait naturellement si on se laisse guider par la preuve. Par contre la démarche inverse semble beaucoup plus délicate.

3.2. Programme

Deux résultats arithmétiques utilisés dans la suite :

preuve de : $e+n \leq m$ *et* $m \leq 2e \Rightarrow 2n \leq m$.

```
let CALCUL1 = "[n:Nat][m:Nat][e:Nat]
        (LE (add e n) m)->(LE m (mult2 e))->(LE (mult2 n) m)";;
let LEM_ARITH1 = "[n:Nat][m:Nat][e:Nat]
            [h1:(LE (add e n) m)][h2:(LE m (mult2 e))]
            (trans_LE (mult2 n) (add e n) m
            (stab_add_r n e n
            (simpl_add_l e n e
            (trans_LE (add e n) m (mult2 e) h1 h2))) h1)";;
LET 'lem_arith1' LEM_ARITH1 CALCUL1;;
```

Preuve de $n \leq m+1$ *et* $e+m \leq t \Rightarrow e+n \leq t+1$.

```
let CALCUL2 = "[n:Nat][m:Nat][e:Nat][t:Nat]
        (LE n (S m))->(LE (add e m) t)->(LE (add e n) (S t))";;
let LEM_ARITH2 = "[n:Nat][m:Nat][e:Nat][t:Nat][h1:(LE n (S m))]
            [h2:(LE (add e m) t)]
            (trans_LE (add e n) (add e (S m)) (S t)
            (stab_add_l n (S m) e h1)
            (add_n_Sm e m ([w:Nat](LE w (S t)))
            (S_LEnm (add e m) t h2)))";;
LET 'lem_arith2' LEM_ARITH2 CALCUL2;;
```

Environnement : On travaille sur un ensemble non vide, cand, sur lequel est définie une relation d'équivalence eg supposée décidable.

L'ensemble des candidats :

```
DECL 'cand' "*";;
```

cand est supposé non vide.

```
AXIOM 'novoid_c' "!C.(cand->C)->C";;
```

Relation d'équivalence

```
DECL 'eg' "[x:cand][y:cand]*";;
```

eg est supposée réflexive, symétrique et transitive :

```
AXIOM 'ref_eg' "(refl cand eg)";;
AXIOM 'sym_eg' "(syme cand eg)";;
AXIOM 'trans_eg' "(trans cand eg)";;
```

eg est supposée décidable.

```
AXIOM 'eg_dec' "[x:cand][y:cand](eg x y)+(~(eg x y))";;
```

Une propriété de eg : (eg x y) et ~(eg y z) ⟶ ~(eg x z).

```
let NO_TRANS = "[x:cand][y:cand][z:cand][h1:(eg x y)][h2:~(eg y z)]
        (eg_dec x z ~(eg x z)
        ([h3:(eg x z)]
        (h2 (trans_eg y x z (sym_eg x y h1) h3) ~(eg x z)))
        <~(eg x z)>Id)";;
LET 'no_trans' NO_TRANS "[x:cand][y:cand][z:cand]
                (eg x y)->(~(eg y z))->(~(eg x z))";;
```

Une liste d'abréviations utiles : les fonctions sur les listes dans le cas de listes d'éléments de type cand.

```
PROP 'list_c' "(list cand)";;
LET 'nil_c' "(nil cand)" "list_c";;
LET 'cons_c' "(cons cand)" "cand->list_c->list_c";;
LET 'length_c' "(length cand)" "list_c->Nat";;
```

Toutes les listes de candidats sont construites :

```
LET 'cons_list' "(axi_list cand)" "(pred_list cand)";;
```

DEFINITION ET PROPRIETES DU NOMBRE DE VOIX D'UN CANDIDAT DANS UNE LISTE.

Définition :

```
let NB_VOIX = "[l:(list_c)][x:cand]
        (l Nat ([y:cand][n:Nat](eg_dec x y Nat
                ([h1:(eg x y)](S n))
                ([h2:~(eg x y)] n))) 0)";;
LET 'nb_voix' NB_VOIX "list_c->cand->Nat";;
```

Le calcul de (nb_voix y.l x) donne après β-réduction :
(eg_dec x y Nat ([h1:(eg x y)](S (nb_voix l x)))
([h2:~(eg x y)](nb_voix l x)))
on va montrer que :
(eg x y)⇒(nb_voix y.l x) = S(nb_voix l x) et
~(eg x y)⇒(nb_voix y.l x) = (nb_voix l x)
Ce résultat nécéssite l'usage du prédicat de construction de la somme disjointe. En effet, si on remplace (eg_dec x y) par Inl(eg x y) ou par Inr ~(eg x y) le résultat découle d'une simple β-réduction ou d'une situation absurde. Ce type de raisonnement est effectué de manière systématique dans l'appendice. Preuve de (eg x y) ⇒ S(nb_voix l x)=(nb_voix y.l x) :

```
let CAS_EG = "[y:cand][l:list_c][x:cand]
        (case1 (eg x y) ~(eg x y) (eg_dec x y)
        ([u:(eg x y)][v:~(eg x y)](v u)) list_c
        (S (nb_voix l) x) (nb_voix l x)))";;
LET 'cas_eg' CAS_EG "[y:cand][l:list_c][x:cand][h:(eg x y)]
            <Nat>(S(nb_voix l x))=(nb_voix (cons_c y l) x)";;
```

Preuve de ~(eg x y) ⇒ (nb_voix l x)=(nb_voix y.l x) :

```
let CAS_DIF = "[y:cand][l:list_c][x:cand]
        (case2 (eg x y) ~(eg x y) (eg_dec x y)
        ([u:(eg x y)][v:~(eg x y)](v u)) list_c
        (S (nb_voix l) x) (nb_voix l x)))";;
LET 'cas_dif' CAS_DIF "[y:cand][l:list_c][x:cand][h:~(eg x y)]
            <Nat>(nb_voix l x)=(nb_voix (cons_c y l) x)";;
```

On en déduit (nb_voix y.l x) ≤ (nb_voix l x) :

```
PROP 'NB_S_PROP' "[y:cand][l:list_c][x:cand]
        (LE (nb_voix (cons_c y l) x) (S (nb_voix l x)))";;

let NB_S = "[y:cand][l:list_c][x:cand]
        (eg_dec x y (NB_S_PROP y l x)
        ([x1:(eg x y)](cas_eg y l x x1 ([w:Nat](LE w (S (nb_voix l x))))
        (LE_n_n (S (nb_voix l x)))))
        ([x2:~(eg x y)]
        (cas_dif y l x x2 ([w:Nat](LE w (S (nb_voix l x))))
        (LE_n_Sn (nb_voix l x)))))";;
LET 'nb_S' NB_S "NB_S_PROP";;
```

Autre conséquence : (eg x y) ⇒ (nb_voix l x)=(nb_voix l y).

```
PROP 'nb_eg_prop' "[x:cand][y:cand][l:list_c]
            <Nat>(nb_voix l x)=(nb_voix l y)";;
```

```
let NB_EG = "[x:cand][y:cand][h:(eg x y)][l:list_c]
        (cons_list l (nb_eg_prop x y) (re Nat 0)
        ([z:cand][l1:list_c][hyp:(nb_eg_prop x y l1)]
        (eg_dec y z (nb_eg_prop x y (cons_c z l1))
        ([t1:(eg y z)]
        (cas_eg z l1 y t1 ([n:Nat]<Nat>(nb_voix (cons_c z l1) x)=n)
            (cas_eg z l1 x (trans_eg x y z h t1)
            ([n:Nat]<Nat>n=(S (nb_voix l1 y)))
                (hyp ([n:Nat]<Nat>(S(nb_voix l1 x))=(S n))
                (re Nat (S (nb_voix l1 x)))))))
        ([t2:~(eg y z)]
        (cas_dif z l1 y t2 ([n:Nat]<Nat>(nb_voix (cons_c z l1) x)=n)
            (cas_dif z l1 x (no_trans x y z h t2)
            ([n:Nat]<Nat>n=(nb_voix l1 y)) hyp))))))";;
LET 'nb_eg' NB_EG "[x:cand][y:cand](eg x y)->(nb_eg_prop x y)";;
```

DEFINITION DE CONSTANTES

(major l x) exprime que x a la majorité dans la liste l, soit : longueur(l) < 2(nb_voix l x).

```
let MAJOR = "[l:list_c][x:cand](LT (length_c l)(mult2 (nb_voix l x)))";;
PROP 'major' MAJOR ;;
```

(no_major l x) exprime la situation contraire .

```
let NO_MAJOR = "[l:list_c][x:cand](LE (mult2 (nb_voix l x)) (length_c l))";;
PROP 'no_major' NO_MAJOR;;
```

*SPECIFICATION DU PROGRAMME : **soit il existe un candidat majoritaire, soit aucun candidat n'est majoritaire.***

```
PROP 'specif' "[l:list_c]<cand>Sig((major l))+(no_major l)";;
```

PREMIERE REDUCTION DU PROBLEME

On va chercher un candidat "possible" c'est-à-dire que s'il n'a pas la majorité, aucun autre candidat ne l'a.

Définition

```
let POSS_MAJ = "[l:list_c][x:cand](no_major l x)->[y:cand](no_major l y)";;
PROP 'poss_maj' POSS_MAJ;;
```

Lorsqu'on a un candidat possible, pour savoir s'il est majoritaire il suffit de compter son nombre de voix. Ceci s'exprime par la proposition suivante :

```
let COMPTE = "[l:list_c][x:cand]
        (poss_maj l x)->((major l x)+(no_major l))";;
PROP 'comptè COMPTE;;
```

Preuve de cette proposition :

```
let MAJ_COMPT = "[l:list_c][x:cand][h:(poss_maj l x)]
        (LE_ou_LT (mult2 (nb_voix l x)) (length_c l)
        (compte l x h)
        ([f1:(no_major l x)]
        (Inr (major l x) (no_major l) (h f1)))
        ([f2:(major l x)]
        (Inl (major l x) (no_major l) f2)))";;
LET 'maj_compt' MAJ_COMPT "compte";;
```

On se ramène à montrer qu'il existe un candidat possible. Ce qui suit est la preuve formelle que l'existence de ce candidat ainsi qu'une preuve de (compte l) donne une preuve de la spécification.

```
let RED1 = "[l:list_c]<cand>Sig((poss_maj l))&(compte l)";;
PROP 'red1' RED1;;

let RED1_SPEC = "[h:red1][l:list_c](h l (specif l)
        ([x1:<cand>Sig((poss_maj l))][x2:(compte l)]
        (x1 (specif l)
        ([x:cand][f:(poss_maj l x)]
        (x2 x f (specif l)
        ([g:(major l x)]
        (Inl (<cand>Sig((major l))) (no_major l)
            (exist cand (major l) x g)))
        (Inr (<cand>Sig((major l))) (no_major l)))))))";;
LET 'red1_spec' RED1_SPEC "red1->specif";;
```

DEUXIEME REDUCTION DU PROBLEME:

***Il reste à montrer** [l:list_c]<cand>Sig((poss-maj l)). On montre en fait un peu plus. On introduit un entier qui contient une estimation du nombre de voix du candidat majoritaire possible de la liste . On construit par récurrence sur la liste un couple formé du candidat retenu à cette étape et d'un entier permettant de changer de candidat possible, qui est la moitié du nombre de personnes à terre plus le nombre de personnes debout, c'est-à-dire le nombre de voix du candidat possible dans la bataille équivalente.*

Abréviations pour les deux projections correspondant à ce couple :

```
LET 'estim' "[c:Nat&cand]<Nat,cand>fst(c)" "(Nat&cand)->Nat";;
LET 'poss_c' "[c:Nat&cand]<Nat,cand>snd(c)" "(Nat&cand)->cand";;
```

On impose trois conditions au couple (e,z) formé :
- *e majore le nombre de voix de z,*
- *les autres candidats n'ont pas plus de longueur(l) - e voix,*
- *si de plus longueur(l) ≤ 2e, cette deuxième condition entraine que les autres candidats ne sont pas majoritaires.*

(nb_voix l z) ≤ e :

```
PROP 'cond1' "[l:list_c][c:Nat&cand](LE (nb_voix l (poss_c c)) (estim c))";;
```

Abréviation de e+(nb_voix l y) ≤ longueur(l) :

```
let PART2= "[l:list_c][c:Nat&cand][y:cand]
        (LE (add (estim c) (nb_voix l y))(length_c l))";;
PROP 'part2' PART2;;
```

(eg x y) ou e+(nb_voix l y) ≤ longueur(l) :

```
let COND2 = "[l:list_c][c:Nat&cand][y:cand](eg (poss_c c) y)+(part2 l c y)";;
PROP 'cond2' COND2;;
```

longueur(l) ≤ 2e :

```
PROP 'cond3' "[l:list_c][c:Nat&cand](LE (length_c l) (mult2 (estim c)))";;
```

Conjonction des trois propriétés :

```
PROP 'cond' "[l:list_c][c:Nat&cand](cond1 l c)&((cond2 l c)&(cond3 l c))";;
```

Abréviations pour la manipulation des conditions : les trois projections ainsi que la formation du triplet.

```
let C1 = "[l:list_c][c:Nat&cand][h:(cond l c)]
      <(cond1 l c),(cond2 l c)&(cond3 l c)>fst(h)";;
LET 'C1' C1 "[l:list_c][c:Nat&cand](cond l c)->(cond1 l c)";;

let C2 = "[l:list_c][c:Nat&cand][h:(cond l c)]
  <(cond2 l c),(cond3 l c)>fst(<(cond1 l c),(cond2 l c)&(cond3 l c)>snd(h))";;
LET 'C2' C2 "[l:list_c][c:Nat&cand](cond l c)->(cond2 l c)";;

let C3 = "[l:list_c][c:Nat&cand][h:(cond l c)]
  <(cond2 l c),(cond3 l c)>snd(<(cond1 l c),(cond2 l c)&(cond3 l c)>snd(h))";
LET 'C3' C3 "[l:list_c][c:Nat&cand](cond l c)->(cond3 l c)";;

let C123 = "[l:list_c][c:Nat&cand]
       [h1:(cond1 l c)][h2:(cond2 l c)][h3:(cond3 l c)]
       let inter = <(cond2 l c),(cond3 l c)>(h2,h3) in
       <(cond1 l c),(cond2 l c)&(cond3 l c)>(h1,inter)";;
LET 'C123' C123 "[l:list_c][c:Nat&cand]
            (cond1 l c)->(cond2 l c)->(cond3 l c)->(cond l c)";;
```

On va montrer l'existence d'un couple vérifiant les trois conditions, c'est-à-dire la proposition suivante :

```
let RED2 = "[l:list_c]<Nat&cand>Sig((cond l))";;
PROP 'red2' RED2;;
```

Cas de la liste vide : le couple qui convient est formé de 0 et de n'importe quel candidat (utilisation de l'hypothèse cand non vide).

```
let C_NIL = "[z:cand]<Nat,cand>(0,z)";;
LET 'c_nil' C_NIL "[z:cand](Nat&cand)";;

let RED_NIL = "let h13 = (LE_n_n 0) in
         let h2 = [z:cand][y:cand]
    (Inr (eg (poss_c (c_nil z)) y) (part2 nil_c (c_nil z) y) (LE_n_n 0)) in
    (novoid_c (red2 nil_c)
          ([z:cand](exist Nat&cand (cond nil_c) (c_nil z)
               (C123 nil_c (c_nil z) h13 (h2 z) h13))))";;
LET 'red_nil' RED_NIL "(red2 nil_c)";;
```

Pas de récurrence. On fait une étude par cas : on a construit le couple (e,z) pour la liste l et on cherche le couple qui convient pour x.l
si x=z on prend (e+1,z) sinon si longueur(l) < 2e on garde (e,z)
et si longueur(l) = 2e on prend (e+1,x).

CAS1 : $x = z$.

Le couple qui convient :

```
let SUIV = "[c:Nat&cand]<Nat,cand>((S (estim c)),(poss_c c))";;
LET 'suiv' SUIV "Nat&cand->Nat&cand";;
```

On montre que ce couple vérifie les conditions :

```
let LEM11 = "[x:cand][l:list_c][c:Nat&cand][f:(eg (poss_c c) x)]
        [h:(cond1 l c)]
        (cas_eg x l (poss_c c) f ([v:Nat](LE v (estim (suiv c))))
            (S_LEnm (nb_voix l (poss_c c)) (estim c) h))";;
LET 'lem11' LEM11 "[x:cand][l:list_c][c:Nat&cand][f:(eg (poss_c c) x)]
            (cond1 l c)->(cond1 (cons_c x l) (suiv c))";;

let LEM12 = "[x:cand][l:list_c][c:Nat&cand][y:cand][h1:~(eg y x)]
        [h2:(part2 l c y)]
        (cas_dif x l y h1
        ([u:Nat](LE (add (S (estim c)) u)(length_c (cons_c x l))))
        (S_LEnm (add (estim c) (nb_voix l y)) (length_c l) h2))";;
let LEM12_PROP = "[x:cand][l:list_c][c:Nat&cand][y:cand]
            ~(eg y x)->(part2 l c y)->(part2 (cons_c x l) (suiv c) y)";;
LET 'lem12' LEM12 LEM12_PROP;;

let LEM12_BIS = "[x:cand][l:list_c][c:Nat&cand][h:(cond2 l c)]
        [f:(eg (poss_c c) x)][y:cand]
        (eg_dec y x (cond2 (cons_c x l) (suiv c) y)
        ([t1:(eg y x)]
        (Inl (eg (poss_c (suiv c)) y) (part2 (cons_c x l) (suiv c) y)
            (trans_eg (poss_c c) x y f (sym_eg y x t1))))
        ([t2:~(eg y x)]
        (h y (cond2 (cons_c x l) (suiv c) y)
        ([t21:(eg (poss_c c) y)]
        (t2 (trans_eg y (poss_c c) x (sym_eg (poss_c c) y t21) f)
            (cond2 (cons_c x l) (suiv c) y)))
        ([t22:(part2 l c y)]
        (Inr (eg (poss_c (suiv c)) y) (part2 (cons_c x l) (suiv c) y)
        (lem12 x l c y t2 t22))))))";;

let LEM12B_PROP = "[x:cand][l:list_c][c:Nat&cand]
        (cond2 l c)->(eg (poss_c c) x)->(cond2 (cons_c x l) (suiv c))";;
LET 'lem12_bis' LEM12_BIS LEM12B_PROP;;
```

On utilise mult2_S, preuve de 2S(n)=S(S(n)).

```
let LEM13 = "[x:cand][l:list_c][e:Nat][h:(LE (length_c l) (mult2 e))]
        (mult2_S e ([v:Nat](LE (length_c (cons_c x l)) v))
        (trans_LE (length_c (cons_c x l)) (S (mult2 e)) (S (S (mult2 e)))
        (S_LEnm (length_c l) (mult2 e) h) (LE_n_Sn (S (mult2 e)))))";;
let LEM13_PROP = "[x:cand][l:list_c][e:Nat]
  (LE (length_c l) (mult2 e))->(LE (length_c (cons_c x l)) (mult2 (S e)))";;
LET 'lem13' LEM13 LEM13_PROP;;
```

On fait la synthèse des lemmes .

```
let CONS_EG = "[x:cand][l:list_c][c:Nat&cand]
        [h:(cond l c)][f:(eg (poss_c c) x)]
        (exist Nat&cand (cond (cons_c x l)) (suiv c)
        (C123 (cons_c x l) (suiv c)
                (lem11 x l c f (C1 l c h))
                (lem12_bis x l c (C2 l c h) f)
                (lem13 x l (estim c) (C3 l c h))))";;

LET 'cons_eg' CONS_EG "[x:cand][l:list_c][c:Nat&cand][h:(cond l c)]
                (eg (poss_c c) x)->(red2 (cons_c x l))";;
```

CAS 2 : ~(eg x z) et longueur(l) < 2e (cette dernière condition est en fait (cond3 x.l c)). *On conserve le même couple c .*

```
let LEM22 = "[x:cand][l:list_c][c:Nat&cand][h:(cond2 l c)][y:cand]
        (h y (cond2 (cons_c x l) c y)
        (Inl (eg (poss_c c) y) (part2 (cons_c x l) c y))
        ([f:(part2 l c y)]
        (Inr (eg (poss_c c) y) (part2 (cons_c x l) c y)
        (lem_arith2
        (nb_voix (cons_c x l) y) (nb_voix l y) (estim c) (length_c l)
        (nb_S x l y) f))))";;
LET 'lem22' LEM22 "[x:cand][l:list_c][c:Nat&cand]
                (cond2 l c)->(cond2 (cons_c x l) c)";;
```

Preuve générale dans ce cas :

```
let CONS_no_EG1 = "[x:cand][l:list_c][c:Nat&cand][h:(cond l c)]
        [f:~(eg (poss_c c) x)][g:(cond3 (cons_c x l) c)]
        (exist Nat&cand (cond (cons_c x l)) c
        (C123 (cons_c x l) c
        (cas_dif x l (poss_c c) f
        ([u:Nat](LE u (estim c)))(C1 l c h))
        (lem22 x l c (C2 l c h)) g))";;

LET 'cons_no_eg1' CONS_no_EG1
      "[x:cand][l:list_c][c:Nat&cand][h:(cond l c)]
       ~(eg (poss_c c) x)->(cond3 (cons_c x l) c)->(red2 (cons_c x l))";;
```

Un résultat intermédiaire : tous les candidats ont un nombre de voix inférieur à l'estimation.

```
let LE_NB_PROP = "[l:list_c][c:Nat&cand][h:(cond l c)][y:cand]
        (LE (nb_voix l y) (estim c))";;

let LE_NB = "[l:list_c][c:Nat&cand][h:(cond l c)][y:cand]
        (C2 l c h y (LE (nb_voix l y) (estim c))
        ([x1:(eg (poss_c c) y)]
        (nb_eg (poss_c c) y x1 l ([n:Nat](LE n (estim c))) (C1 l c  h)))
        ([x2:(part2 l c y)]
        (simpl_add_l (estim c) (nb_voix l y) (estim c)
        (trans_LE (add (estim c) (nb_voix l y)) (length_c l)
        (mult2 (estim c)) x2 (C3 l c h)))))";;
LET 'LE_nb' LE_NB LE_NB_PROP;;
```

CAS 3 : ~(eg z x) et longueur(l)=2e . On prend alors le couple (S(e),x).

```
let CHANG = "[c:Nat&cand][x:cand]<Nat,cand>((S (estim c)),x)";;
LET 'chang' CHANG "Nat&cand->cand->Nat&cand";;

let LEM32_PROP = "[x:cand][l:list_c][c:Nat&cand]
          [g:<Nat>(mult2 (estim c))=(length_c l)]
          (cond l c)->(cond2 (cons_c x l) (chang c x))";;

let LEM32 = "[x:cand][l:list_c][c:Nat&cand]
        [g:<Nat>(mult2 (estim c))=(length_c l)]
        [h:(cond l c)][y:cand]
        (eg_dec y x (cond2 (cons_c x l) (chang c x) y)
        ([x1:(eg y x)]
        (Inl (eg (poss_c (chang c x)) y)
        (part2 (cons_c x l) (chang c x) y)(sym_eg y x x1)))
        ([x2:~(eg y x)](Inr (eg (poss_c (chang c x)) y)
        (part2 (cons_c x l) (chang c x) y)
        (cas_dif x l y x2
        ([v:Nat](LE (add (estim (chang c x)) v)(length_c (cons_c x l))))
        (S_LEnm (add (estim c) (nb_voix l y)) (length_c l)
        (g ([v:Nat](LE (add (estim c) (nb_voix l y)) v))
        (stab_add_l (nb_voix l y) (estim c) (estim c)
        (LE_nb l c h y))))))))";;
LET 'lem32' LEM32 LEM32_PROP;;

let CONS_no_EG2_PROP = "[x:cand][l:list_c][c:Nat&cand]
            [g:<Nat>(mult2 (estim c))=(length_c l)]
            (cond l c)->(red2 (cons_c x l))";;

let CONS_no_EG2 = "[x:cand][l:list_c][c:Nat&cand]
          [g:<Nat>(mult2 (estim c))=(length_c l)]
          [h:(cond l c)]
          (exist Nat&cand (cond (cons_c x l)) (chang c x)
          (C123 (cons_c x l) (chang c x)
          (cas_eg x l x (ref_eg x) ([v:Nat](LE v (estim (chang c x))))
          (S_LEnm (nb_voix l x) (estim c) (LE_nb l c h x)))
          (lem32 x l c g h)
          (lem13 x l (estim c) (C3 l c h))))";;
LET 'cons_no_eg2' CONS_no_EG2 CONS_no_EG2_PROP;;
```

Pas de récurrence : synthèse des trois cas précédents.

```
let RED2_CONS = "[x:cand][l:list_c][hyp:(red2 l)]
          (hyp (red2 (cons_c x l))
          ([c:Nat&cand][h:(cond l c)]
          (eg_dec (poss_c c) x (red2 (cons_c x l))
          (cons_eg x l c h)
          ([f:~(eg (poss_c c) x)]
          (LE_EGLT (length_c l) (mult2 (estim c)) (C3 l c h)
          (red2 (cons_c x l))
          ([g1:<Nat>(length_c l)=(mult2 (estim c))]
          (cons_no_eg2 x l c
          (sym Nat (length_c l)(mult2 (estim c)) g1) h))
          ([g2:(LT (length_c l) (mult2 (estim c)))]
          (cons_no_eg1 x l c h f g2)))))))";;
LET 'red_cons' RED2_CONS "[x:cand][l:list_c](red2 l)->(red2 (cons_c x l))";;
```

PREUVE DE RED2

```
let RED_PROOF = "[l:list_c](cons_list l red2 red_nil red_cons)";;
LET 'red_proof' RED_PROOF "red2";;
```

On va prouver maintenant que ce résultat permet d'obtenir le candidat possible. On montre d'abord que les conditions 2 et 3 impliquent que les candidats autres que celui retenu ne sont pas majoritaires.

```
let LEMME = "[l:list_c][c:Nat&cand][h:(cond2 l c)&(cond3 l c)][y:cand]
        (h (eg (poss_c c) y)+(no_major l y)
        ([h1:(cond2 l c)][h2:(cond3 l c)]
        (h1 y (eg (poss_c c) y)+(no_major l y)
        (Inl (eg (poss_c c) y)(no_major l y))
        ([h12:(LE (add (estim c) (nb_voix l y))(length_c l))]
        (Inr (eg (poss_c c) y)(no_major l y)
        (lem_arith1 (nb_voix l y)(length_c l)(estim c) h12 h2))))))";;

LET 'lemmè LEMME "[l:list_c][c:Nat&cand][h:(cond2 l c)&(cond3 l c)]
            [y:cand](eg (poss_c c) y)+(no_major l y)";;

let RED2_POSSEX = "[h:red2][l:list_c]
          (h l <cand>Sig((poss_maj l))
          ([c:Nat&cand][f:(cond l c)]
          (exist cand (poss_maj l) (poss_c c)
          ([g:(no_major l (poss_c c))]
          [z:cand]
          (lemme l c <(cond1 l c),((cond2 l c)&(cond3 l c))>snd(f) z
          (no_major l z)
          ([t:(eg (poss_c c) z)]
          (nb_eg (poss_c c) z t l
             ([v:Nat](LE (mult2 v) (length_c l))) g))
          <(no_major l z)>Id)))))";;

LET 'red2_poss_ex' RED2_POSSEX "red2->[l:list_c]<cand>Sig((poss_maj l))";;
```

On reconstitue alors la preuve de la spécification initiale

```
let PROGRAM = "(red1_spec ([l:list_c]
        let poss_ex = (red2_poss_ex red_proof l) in
      <<cand>Sig((poss_maj l)),(compte l)>(poss_ex,(maj_compt l))))";;
LET 'program' PROGRAM "specif";;
```

3.3. Remarques

Le but de cette étude était le développement d'un programme plus important que les précédents.

Commençons par examiner la structure de la preuve. Elle suit de près le principe de l'algorithme. La spécification générale est celle d'un programme quelconque de recherche de majorité. La première spécification n'impose l'utilisation que d'un seul compteur : on sélectionne un candidat (le seul possible) et on sait ensuite déterminer s'il a ou non la majorité. La seconde réduction consiste à construire effectivement le candidat possible en modélisant la "bagarre" à l'aide de l'estimateur e.

Quelques améliorations : on a remplacé l'égalité sur les candidats par une relation d'équivalence. On peut aussi se demander s'il est nécessaire de recompter les voix après coup. Dans le cas où l=2e , il n'y a plus de combattant et on montre simplement que le candidat retenu n'a pas la majorité. On pourrait donc améliorer la preuve par un test à la suite duquel on pourrait éventuellement passer de la preuve de ***red2*** à la spécification. Par contre les listes (*ABC*) et (*ABB*) sont un exemple de la nécessité de recompter. En effet, à la fin de la bagarre, les vainqueurs sont respectivement *C* et *B* et les deux compteurs indiquent 2.

Cet exemple nous amène à faire quelques nouvelles remarques.

Ici on construit de manière "simple" la fonction ***nb_voix*** de type $cand \rightarrow (list\ cand) \rightarrow Nat$. Il faut ensuite prouver un certain nombre de propriétés de ce programme. Par β-réduction on connait le comportement de $(nb\_voix\ nil\ x)$. Par contre l'information sur $(nb\_voix\ y.l\ x)$ n'est pas suffisante, il faut aller plus loin dans la structure du programme et utiliser la structure de $(eg\_dec\ x\ y)$. Une fois connus $(nb\_voix\ y.l\ x)$ et $(nb\_voix\ nil\ x)$. Les résultats sur ***nb_voix*** se prouvent par induction sur la liste, c'est à dire par application du principe de récurrence sur les listes ***axi_list***.

La structure de donnée choisie pour représenter les résultats du vote par Boyer et Moore ou par R.Backhouse est un tableau. Nous avons donc commencé par essayer de représenter un tableau en constructions. On pourrait le voir comme une application a de $[0..n]$ dans N c'est-à-dire que l'on poserait :
$array = !A.[n:Nat][p:Nat](LE\ p\ n) \rightarrow A$,
mais ceci est peu commode car à chaque accès à $a[i]$ il faut exhiber une preuve de $i \leq n$. En fait il y a une difficulté à manipuler des fonctions partielles. Dans un langage de programmation, le test est réalisé mais cela devient très lourd dans une preuve particulière car il faut alors gérer l'erreur de manière explicite. Toutes les fonctions, par conséquent, doivent être construites sur une somme disjointe d'une proposition et d'un ensemble d'erreurs. Il faut donc commencer par un test. Dans un cas d'erreur la fonction déterminera s'il faut propager celle-ci ou bien la traiter. Il est intéressant de pouvoir modèliser de tels phénomènes. Cependant nous avons

préféré une structure de liste qui rendait la preuve plus lisible. De plus la structure du programme est bien adaptée à celle de liste puisqu'en fait on ne regarde à chaque fois que le dernier élément.

On aurait pu voir les tableaux sous forme d'arbres de stockage. C'est une version améliorée des listes d'un point de vue temps d'accès, mais cela ne change en rien la preuve.

Nous allons développer la construction d'arbre de stockage dans l'unique but de montrer que l'on code aisément des structures un peu complexes.

Arbres

Définition générale

```
let TREE = "!A.[f:A->A->A][g:A]A";;
PROP 'treè TREE;;
```

Définition d'arbres avec feuilles étiquetées :

```
let STO_TREE = "!A,B.[f:B->B->B][g:B][h:A->B]B";;
PROP 'sto_treè STO_TREE;;
```

Feuilles de ces arbres :

```
let FEUILLE = "!A.[f:A->A->A][g:A]g";;
LET 'feuillè FEUILLE "tree";;

let FEUI_VIDE ="!A,B.[f:B->B->B][g:B][h:A->B]g";;
LET 'feui_vidè FEUI_VIDE "sto_tree";;

let FEUI_PLEIN ="!A.[x:A]!B.[f:B->B->B][g:B][h:A->B](h x)";;
LET 'feui_plein' FEUI_PLEIN "!A.A->(sto_tree A)";;
```

Formation de l'arbre père de deux arbres :

```
let PERE= "[h1:tree][h2:tree]!A.[f:A->A->A][g:A](f (h1 A f g) (h2 A f g))";;
LET 'perè PERE "tree->tree->tree";;

let STO_PERE = "!A.[h1:(sto_tree A)][h2:(sto_tree A)]!B.[f:B->B->B][g:B]
        [h:A->B](f (h1 B f g h) (h2 B f g h))";;
LET 'sto_perè STO_PERE "!A.(sto_tree A)->(sto_tree A)->(sto_tree A)";;
```

Un arbre général est aussi un arbre étiqueté.

```
let INJ_STO = "[h:tree]!A.(h (sto_tree A) (sto_pere A) (feui_vide A))";;
LET 'inj_sto' INJ_STO "tree->sto_tree";;
```

Hauteur d'un arbre :

```
let HAUT = "!A.[h:(sto_tree A)](h Nat max 0 ([x:A]0))";;
LET 'haut' HAUT "!A.(sto_tree A)->Nat";;
```

Prédicat qui détermine si un arbre étiqueté est plein :

```
let PLEIN = "!A.[h:(sto_tree A)](h bool et false ([x:A] true))";;
LET 'plein' PLEIN "!A.(sto_tree A)->bool";;
```

Prédicats de constructions pour les arbres : ils indiquent qu'un arbre est soit une feuille soit de la forme (pere h1 h2) :

```
let PRED_TREE = "[h:tree]{P|tree->*}[base:(P feuille)]
          [rec:[h1:tree][h2:tree](P h1)->(P h2)->(P (pere h1 h2))]
          (P h)";;
PROP 'pred_treè PRED_TREE;;
```

Quelques arbres construits :

```
let PRED_FEUI = "{P|tree->*}[base:(P feuille)]
          [rec:[h1:tree][h2:tree](P h1)->(P h2)->(P (pere h1 h2))]
          base";;
LET 'pred_feui' PRED_FEUI "(pred_tree feuille)";;

let PRED_PERE = "[h1:tree][h2:tree][f1:(pred_tree h1)][f2:(pred_tree h2)]
          {P|tree->*}[base:(P feuille)]
          [rec:[g1:tree][g2:tree](P g1)->(P g2)->(P (pere g1 g2))]
          (rec h1 h2 (f1 P base rec)(f2 P base rec))";;
LET 'pred_perè PRED_PERE "[h1:tree][h2:tree]
          (pred_tree h1)->(pred_tree h2)->(pred_tree (pere h1 h2))";;
```

Prédicat de construction pour les arbres étiquetés :

```
let PRED_STO = "!A.[h:(sto_tree A)]{P|(sto_tree A)->*}
          [vide:(P (feui_vide A))][pleine :[x:A](P (feui_plein A x))]
           [rec:[h1:(sto_tree A)][h2:(sto_tree A)]
                        (P h1)->(P h2)->(P (sto_pere A h1 h2))]
          (P h)";;
PROP 'pred_sto' PRED_STO;;
```

Arbres binaires équilibrés définis en fonction de leur hauteur :

```
let BIN = "[n:Nat](n tree ([h:tree](pere h h)) feuille)";;
LET 'bin' BIN "Nat->tree";;
```

On prouve par récurrence sur leur hauteur qu'ils sont construits.

```
let PRED_BIN = "[n:Nat](peano n ([u:Nat](pred_tree (bin u)))
          (pred_feui)([u:Nat][h:(pred_tree (bin u))]
          (pred_pere (bin u) (bin u) h h)))";;
LET 'pred_bin' PRED_BIN "[n:Nat](pred_tree (bin n))";;
```

INSERTION

Insertion d'un élément dans un arbre général construit, à la place de la feuille la plus à gauche.

```
let INS_TREE = "!A.[x:A][h:tree][cons:(pred_tree h)]
          (cons (tree->(sto_tree A)) (feui_plein A x)
          ([h1:tree][h2:tree][i1:(sto_tree A)][i2:(sto_tree A)]
          (sto_pere A i1 (inj_sto h2 A))))";;
LET 'ins_treè INS_TREE "!A.A->[h:tree](pred_tree h)->(sto_tree A)";;
```

Insertion dans un arbre équilibré :

```
let INS_BIN = "!A.[x:A][n:Nat](ins_tree A x (bin n) (pred_bin n))";;
LET 'ins_bin' INS_BIN "!A.A->[n:Nat](sto_tree A)";;
```

Insertion dans un arbre étiqueté. On insère sur la feuille libre la plus à gauche. Si l'arbre est plein, on forme un nouvel arbre dont le fils gauche est l'arbre initial et le fils droit un arbre binaire équilibré de même hauteur avec l'élément à insérer dans sa feuille la plus à gauche.

```
let INS_STO = "!A.[x:A][h:(sto_tree A)][cons:(pred_sto A h)]
        (cons ((sto_tree A)->(sto_tree A))
        (feui_plein A x)
        ([y:A](sto_pere A (feui_plein A y) (feui_plein A x)))
        ([h1:(sto_tree A)][h2:(sto_tree A)]
        [i1:(sto_tree A)][i2:(sto_tree A)]
        (if (sto_tree A) (plein A h1)
        (if (sto_tree A) (plein A h2)
          (sto_pere A (sto_pere A h1 h2)
                (ins_bin A x (S (haut A h1))))
          (sto_pere A h1 i2))
        (sto_pere A i1 h2))))";;
LET 'ins_sto' INS_STO "!A.[x:A][h:(sto_tree A)]
              (pred_sto A h)->(sto_tree A)";;
```

Il faudrait encore montrer que les arbres obtenus par insertions successives d'éléments sont tous construit ou bien poser comme axiome que tous les *sto_tree* sont construits. Ensuite on travaillerait sur des arbres qui sont soit une feuille vide, soit le produit de l'insertion d'un élément dans un arbre. Evidemment nous avons fait un choix particulier de stratégie d'insertion, il est possible d'en modéliser bien d'autres.

Cet exemple est le premier algorithme un peu complexe que nous avons développé. On remarque que la spécification est assez directe à écrire, il suffit de savoir calculer le nombre de voix d'un candidat dans une liste et de connaitre l'ordre sur les entiers. Les spécifications des exemples qui suivent posaient par contre quelques problèmes.

4. Algorithme de mise en page

4.1. Présentation

Cet exemple, comme celui de la fonction lambo, provient du workshop de Göteborg sur les spécifications. On dispose d'une liste de caractères qui sont soit des lettres, soit un espace (blanc ou retour à la ligne). On peut voir cette liste comme une suite de mots (liste non vide de lettres) séparés par des suites d'espacements. Le but du programme est de former la même liste de mots séparés par seulement un caractère blanc ou retour à la ligne de manière à mettre le maximum de mots par ligne. On qualifiera de "formatté" un texte vérifiant ces propriétés.

La spécification générale de ce problème a la forme:

$$\forall l \in list\ \exists m \in list \cdot (format\ m) \& (equiv\ l\ m)$$

où $(format\ m)$ et $(equiv\ l\ m)$ signifient respectivement que m est formatté et que l et m sont équivalents au sens où ils représentent le même ensemble de mots.

Le principal problême est l'axiomatisation de cette spécification. Comment représenter une ligne, un mot, un texte formatté... On se rend compte qu'une définition prenant en compte les notions de mots et de lignes de manière abstraite n'est pas du tout maniable. On va donc dire qu'un texte formatté est une liste de caractères particulière, c'est-à-dire qu'on limite les possibilités de construction. Une liste de caractères est de manière générale soit *nil* soit le *cons* d'un caractère et d'une autre liste de caractères, un texte formatté est soit *nil*, soit le *cons* d'une lettre et d'un texte formatté dont la dernière ligne n'est pas pleine, soit etc. On a ainsi un certain nombre de moyens de constructions. On doit également définir ce que signifie l'équivalence de deux textes. Informellement c'est l'égalité des listes de mots sous-jacentes. On introduit donc une fonction *norm* sur l'ensemble des listes de caractères telle que $(norm\ l)$ soit la liste formée de tous les mots de l séparés par un blanc. Pour écrire la fonction *norm*, on a besoin de prédicats indiquant si une liste finit par un lettre ou non, puis on définit cette fonction par récurrence (c'est-à-dire en appliquant la liste). De même il n'est pas difficile d'écrire un programme qui donne la dernière ligne d'un texte (i.e. la liste de caractères après le dernier retour à la ligne). Tout ceci correspond à un point de vue très pratique: on a dans un buffer la dernière ligne du programme et quelque part une information sur la nature du dernier caractère du texte formatté que l'on est en-train de former, les caractères à traiter arrivent tour-à-tour, on sait reconnaître si ce sont des lettres ou non et on décide alors que faire.

Nous donnons ensuite le programme. Celui-ci n'a pu être vérifié mécaniquement car le contrôle des types prenait un temps sans doute fini mais dont on ne voyait pas la fin. Ceci est vraiment un problême de cette implémentation (l'une des premières). Le programme tourne par contre dans une nouvelle version. La syntaxe de cette dernière étant différente de ce qui est exposé ici, nous préférons donner le programme primitif. Il y a donc sans-doute dans les lignes qui suivent de petites erreurs facilement corrigeables. Ce qui est important est de toute manière le principe de la résolution.

4.2. Programme

Le cadre de travail :

L'espace des caractères

```
DECL 'ch' "*";;
```

Deux caractères particuliers distincts : le blanc et le retour à la ligne.

```
AXIOM 'bl' "ch";;
AXIOM 'lf' "ch";;
AXIOM 'dif_lf_bl' "~(<ch>lf=bl)";;
```

Les espaces sont les blancs ou les retours à la ligne, les lettres sont les autres caractères.

```
PROP 'sp' "[x:ch](<ch>bl=x)+(<ch>lf=x)";;
PROP 'ltr' "[x:ch](sp x)->{}";;
```

bl et lf sont des espaces :

```
LET 'sp_bl' "(Inl (<ch>bl=bl) (<ch>lf=bl) (re ch bl))" "(sp bl)";;
LET 'sp_lf' "(Inr (<ch>bl=lf) (<ch>lf=lf) (re ch lf))" "(sp lf)";;
```

Etre un espace est une propriété décidable :

```
AXIOM 'ltr_or_sp' "[x:ch](ltr x)+(sp x)";;
```

Abréviations pour la manipulation de listes de caractères :

```
PROP 'l_ch' "(list ch)";;
LET 'nil_ch' "(nil ch)" "l_ch";;
LET 'co_ch' "(cons ch)" "ch->l_ch->l_ch";;
LET 'lgth_ch' "(length ch)" "l_ch->Nat";;
LET 'app_ch' "(append ch)" "l_ch->l_ch->l_ch";;
```

Longueur maximale d'une ligne (sans le caractère de fin de ligne).

```
AXIOM 'max' "Nat";;
```

Prédicats sur les listes :

Terminer par une lettre (cette définition est équivalente à : il existe a et m tels que a soit une lettre et l = a.m). L'utilisation d'une définition avec quantification sur les prédicats est plus souple à l'utilisation.

```
let END_LTR = "[l:l_ch]{P|l_ch->*}
        ([a:ch][m:l_ch][h:(ltr a)](P (co_ch a m)))->(P l)";;
PROP 'end_ltr' END_LTR ;;
```

Etre vide ou terminer par un espace. On montrera que c'est le contraire de "terminer par une lettre".

```
let END_VOID = "[l:l_ch]{P|l_ch->*}
        (P nil_ch)->([a:ch][m:l_ch][h:(sp a)](P (co_ch a m)))->(P l)";;
PROP 'end_void' END_VOID ;;
```

Les propriétés évidentes de constructions de listes vérifiant ces prédicats :

```
LET 'end_ltr_co' "[a:ch][l:l_ch][h:(ltr a)]{P|l_ch->*}
                  [g:[b:ch][m:l_ch][h':(ltr b)](P (co_ch b m))]
                  (g a l h)"
                 "[a:ch][l:l_ch](ltr a)->(end_ltr (co_ch a l))";;

LET 'end_void_nil' "{P|l_ch->*}[h1:(P nil_ch)]
                    [h2:[a:ch][m:l_ch][h:(sp a)](P (co_ch a m))]h1"
                   "(end_void nil_ch)";;

LET 'end_void_co' "[a:ch][l:l_ch][h:(sp a)]{P|l_ch->*}[h1:(P nil_ch)]
                   [g:[b:ch][m:l_ch][h':(sp b)](P (co_ch b m))]
                   (g a l h)"
                  "[a:ch][l:l_ch](sp a)->(end_void (co_ch a l))";;
```

Autres propriétés :

```
LET 'ltr_end_ltr' "[a:ch][l:l_ch][h:(end_ltr (co_ch a l))]
                   (h ([m:l_ch](ltr (hd_tot ch a m)))
                   ([b:ch][m:l_ch][h1:(ltr b)]h1))"
                  "[a:ch][l:l_ch]
                   (end_ltr (co_ch a l))->(ltr a)";;

LET 'sp_end_sp' "[a:ch][l:l_ch][h:(end_sp (co_ch a l))]
                   (h ([m:l_ch](sp (hd_tot ch a m)))
                   ([b:ch][m:l_ch][h1:(sp b)]h1))"
                  "[a:ch][l:l_ch]
                   (end_sp (co_ch a l))->(sp a)";;
```

On montre qu'une liste vérifie soit end_ltr, soit end_void :

```
let END_VOID_LTR = "[l:l_ch](axi_list ch l ([m:l_ch](end_void m)+(end_ltr m))
                   (Inl (end_void nil_ch) (end_ltr nil_ch) end_void_nil)
                   ([a:ch][m:l_ch][h:(end_void m)+(end_ltr m)]
                   (ltr_or_sp a (end_void (co_ch a m))+(end_ltr (co_ch a m))
                   ([h1:(ltr a)]
                   (Inr (end_void (co_ch a m)) (end_ltr (co_ch a m))
                      (end_ltr_co a m h1)))
                   ([h2:(sp a)]
                   (Inl (end_void (co_ch a m)) (end_ltr (co_ch a m))
                      (end_void_co a m h2))))))";;

LET 'end_void_ltr' END_VOID_LTR "[l:l_ch](end_void l)+(end_ltr l)";;
```

Ces deux conditions sont contradictoires :

```
let END_VOID_LTR_ABS = "[l:l_ch][h1:(end_void l)]
      (h1 ([m:l_ch]~(end_ltr m))
         ([f1:(end_ltr nil_ch)]
          (h1 ([m:l_ch]~(<l_ch>nil_ch=m))
              ([a:ch][m:l_ch][t:(ltr a)][f:<l_ch>nil_ch=(co_ch a m)]
             (axi_ari (lgth_ch m)
                 (f ([n:l_ch]<Nat>(lgth_ch n)=0) (re Nat 0))))))
          ([a:ch][m:l_ch][f:(sp a)][h:(end_ltr (co_ch a m))]
              (f (ltr_end_ltr a m h))));;

LET 'end_void_ltr_abs' END_VOID_LTR_ABS
                  "[l:l_ch](end_void l)->(~(end_ltr l))";;
```

AUTOUR DE L'EQUIVALENCE DE DEUX LISTES

Définition de l'équivalence de deux listes de caractères : c'est l'égalité des formes normales des listes, à savoir les mots séparés par un blanc.

Fonction qui agit à l'insertion d'un espace : on met un blanc si la liste finit par une lettre, sinon on renvoit la même liste.

```
LET 'aj_spà "[l:l_ch](end_void_ltr l l_ch
                ([h1:(end_void l)]l)
                ([h2:(end_ltr l)](co_ch bl l)))"
          "l_ch->l_ch";;
```

La fonction normal teste si le caractère est une lettre auquel cas elle l'ajoute, sinon elle appelle aj_spa.

```
let NORMAL = "[l:l_ch](l l_ch ([a:ch][m:l_ch]
                    (ltr_or_sp a l_ch ([h1:(ltr a)](co_ch a m))
                                      ([h2:(sp a)](aj_spa m))))
                    nil_ch)";;
LET 'norm' NORMAL "l_ch->l_ch";;
```

Définition de la notion d'équivalence comme égalité des formes normales.

```
PROP 'equiv' "[l:l_ch][m:l_ch]<l_ch>(norm l)=(norm m)";;
```

On en déduit immédiatement que equiv est une relation d'équivalence :

```
LET 'eq_rè "[l:l_ch](re l_ch (norm l))" "[l:l_ch](equiv l l)";;

LET 'eq_sym' "[l:l_ch][m:l_ch][h1:(equiv l m)]
          (sym l_ch (norm l) (norm m) h1)"
          "[l:l_ch][m:l_ch](equiv l m)->(equiv m l)";;

LET 'eq_tran' "[l:l_ch][m:l_ch][n:l_ch][h1:(equiv l m)][h2:(equiv m n)]
           (tran l_ch (norm l) (norm m) (norm n) h1 h2)"
           "[l:l_ch][m:l_ch][n:l_ch]
           (equiv l m)->(equiv m n)->(equiv l n)";;
```

Pour montrer des propriétés plus fortes sur equiv on raisonne par récurrence sur la liste. Pour cela on a besoin de connaître la valeur de (norm (co_ch a l)) en fonction de (norm l). La preuve est analogue à celle du comportement de nb_voix dans l'algorithme de vote.

Propriétés de aj_spa :

```
let AJ_SP_VOID = "[l:l_ch](case1 (end_void l) (end_ltr l)
              (end_void_ltr_abs l) l_ch l (co_ch bl l)(end_void_ltr l))";;
LET 'aj_sp_void' AJ_SP_VOID "[l:l_ch]
                    (end_void l)-><l_ch>l=(aj_spa l)";;

let AJ_SP_LTR = "[l:l_ch](case2 (end_voi l) (end_ltr l)
              (end_void_ltr_abs l) l_ch l (co_ch bl l)(end_void_ltr l))";;
LET 'aj_sp_ltr' AJ_SP_LTR "[l:l_ch]
                  (end_ltr l)-><l_ch>(co_ch bl l)=(aj_spa l)";;
```

Propriétés de norm :

```
let NORM_LTR = "[a:ch][l:l_ch]
          (case1 (ltr a) (sp a) ([l1:(ltr a)][l2:(ltr a)](l2 l1))
           l_ch (co_ch a (norm l))(aj_spa (norm l))(end_void_ltr a))";;
LET 'norm_ltr' NORM_LTR "[a:ch][l:l_ch]
          (ltr a)->(l_ch)(co_ch a (norm l))=(norm (co_ch a l))";;

let NORM_SP = "[a:ch][l:l_ch]
          (case2 (ltr a) (sp a) ([l1:(ltr a)][l2:(ltr a)](l2 l1))
           l_ch (co_ch a (norm l))(aj_spa (norm l))(end_void_ltr a))";;
LET 'norm_sp' NORM_SP "[a:ch][l:l_ch]
          (sp a)->(l_ch) (aj_spa (norm l))=(norm (co_ch a l))";;

let NORM_SP_LTR = "[a:ch][l:l_ch][h1:(end_ltr (norm l))][h2:(sp a)]
             (tran l_ch (co_ch b1 (norm l))(aj_spa (norm l))
             (norm (co_ch a l))
             (aj_spa_ltr (norm l) h1)
             (norm_sp a l h2))";;
LET 'norm_sp_ltr' NORM_SP_LTR "[a:ch][l:l_ch]
 (end_ltr (norm l))->(sp a)->(l_ch)(co_ch b1 (norm l))=(norm (co_ch a l))";

let NORM_SP_VOID = "[a:ch][l:l_ch][h1:(end_void (norm l))][h2:(sp a)]
             (tran l_ch (norm l) (aj_spa (norm l))
             (norm (co_ch a l))
             (aj_spa_void (norm l) h1)
             (norm_sp a l h2))";;

LET 'norm_sp_void' NORM_SP_VOID "[a:ch][l:l_ch]
  (end_void (norm l))->(sp a)->(l_ch)(norm l)=(norm (co_ch a l))";;
```

D'où des propriétés de la relation equiv :

```
let EQ_LTR = "[a:ch][l:l_ch][m:l_ch][h1:(equiv l m)][h2:(ltr a)]
          (norm_ltr a l h2 ([l':l_ch](l_ch)l'=(norm (co_ch a m)))
          (norm_ltr a m h2 ([m':l_ch](l_ch)(co_ch a (norm l))=m')
          (h1 ([m':l_ch](l_ch)(co_ch a (norm l))=(co_ch a m'))
          (re l_ch (co_ch a (norm l))))))";;

LET 'eq_ltr' EQ_LTR "[a:ch][l:l_ch][m:l_ch]
          (equiv l m)->(ltr a)->(equiv (co_ch a l) (co_ch a m))";;

let EQ_SPA = "[a:ch][l:l_ch][b:ch][m:l_ch][h1:(equiv l m)]
          [h2:(sp a)][h3:(sp b)]
          (end_void_ltr (norm l) (equiv (co_ch a l) (co_ch b m))
          ([g1:(end_void (norm l))]
          (norm_sp_void a l g1 h2 ([p:l_ch](l_ch)p=(norm (co_ch b m)))
          (norm_sp_void b m (h1 ([p:l_ch](end_void p)) g1) h3
          ([p:l_ch](l_ch)(norm l)=p) h1)))
          ([g2:(end_ltr (norm l))]
          (norm_sp_ltr a l g2 h2 ([p:l_ch](l_ch)p=(norm (co_ch b m)))
          (norm_sp_ltr b m (h1 ([p:l_ch](end_ltr p)) g2) h3
          ([p:l_ch](l_ch)(co_ch b1 (norm l))=p)
          (h1 ([p:l_ch](l_ch)(co_ch b1 (norm l))=(co_ch b1 p))
          (re l_ch (co_ch b1 (norm l)))))))";;
```

```
LET 'eq_spà EQ_SPA "[a:ch][l:l_ch][b:ch][m:l_ch]
      (equiv l m)->(sp a)->(sp b)->(equiv (co_ch a l) (co_ch b m))";;

let EQ_CO = "[a:ch][l:l_ch][m:l_ch][h:(equiv l m)]
        (ltr_or_sp a (equiv (co_ch a l) (co_ch a m))
        ([h1:(ltr a)](eq_ltr a l m h h1))
        ([h2:(sp a)](eq_spa a l a m h h2 h2)))";;

LET 'eq_co' EQ_CO  "[a:ch][l:l_ch][m:l_ch]
        (equiv l m)->(equiv (co_ch a l) (co_ch a m))";;

let EQ_APP ="[l:l_ch][m:l_ch][h:(equiv l m)][n:l_ch]
        (pred_ch n ([n':l_ch](equiv (app_ch n' l) (app_ch n' m))) h
        ([a:ch][n':l_ch][h':(equiv (app_ch n' l)(app_ch n' m))]
        (eq_co a (app_ch n' l) (app_ch n' m) h')))";;

LET 'eq_app' EQ_APP "[l:l_ch][m:l_ch][h:(equiv l m)][n:l_ch]
            (equiv (app_ch n l)(app_ch n m))";;
```

LE BUFFER

On définit maintenant la fonction buffer qui renvoit la dernière ligne d'un texte (i.e tous les caractères après le dernier lf).

On montre d'abord qu'un blanc ou une lettre ne sont pas des retours à la ligne.

```
LET 'bl_no_lf' "[c:ch][h:<ch>bl=c][g:<ch>lf=c]
        (dif_lf_bl (tran ch lf c bl g (sym ch bl c h)))"
       "[c:ch](<ch>bl=c)->~(<ch>lf=c)";;

LET 'ltr_no_lf' "[c:ch][h:(ltr c)][g:<ch>lf=c]
        (h (g ([m:ch](sp m)) sp_lf))"
       "[c:ch](ltr c)->~(<ch>lf=c)";;
```

On montre que l'on sait décider si on a un retour à la ligne ou non.

```
let DECID_LF = "[c:ch](ltr_or_sp c (decid <ch>lf=c)
        ([h1:(ltr c)](Inl ~<ch>lf=c <ch>lf=c (ltr_no_lf c h1)))
        ([h2:(sp c)](h2 (decid <ch>lf=c)
            ([h21:<ch>bl=c]
            (Inl ~<ch>lf=c <ch>lf=c (bl_no_lf c h21)))
            ([h22:<ch>lf=c]
            (Inr ~<ch>lf=c <ch>lf=c h22)))))";;

LET 'decid_lf' DECID_LF  "[c:ch](decid <ch>lf=c)";;
```

La fonction buffer : dès que l'on rencontre un "lf", on remet le buffer à nil sinon on ajoute les caractères un par un.

```
let BUFFER = "[l:l_ch](l l_ch ([a:ch][m:l_ch]
                (decid_lf a l_ch
            ([h1:~<ch>lf=a](co_ch a m))
            ([h2:<ch>lf=a]nil_ch))) nil_ch)";;
LET 'buffer' BUFFER "l_ch->l_ch";;
```

Propriétés de la fonction buffer :

```
LET 'buffer_lf' "[c:ch][l:l_ch](case2 ~<ch>lf=c <ch>lf=c
        l_ch (co_ch c (buffer l)) nil_ch (decid_lf c))"
      "[c:ch][l:l_ch]<ch>lf=c->(<l_ch>nil_ch=(buffer (co_ch c l)))";;

LET 'buffer_no_lf' "[c:ch][l:l_ch]
          (case1 ~<ch>lf=c <ch>lf=c
          l_ch (co_ch c (buffer l)) nil_ch (decid_lf c))"
         "[c:ch][l:l_ch]
       ~(<ch>lf=c)->(<l_ch>(co_ch c (buffer l))=(buffer (co_ch c l)))";;
```

SEPARATION DU DERNIER MOT D'UNE LISTE

Définitions de fonctions qui séparent une liste en trois parties : le dernier mot (éventuellement vide si la liste ne commence pas par une lettre), les espaces qui précèdent ce mot et enfin le reste de la liste (soit vide, soit commençant par une lettre).

Abréviations pour manipuler les triplets de listes.

```
PROP 'tri_l_ch' "l_ch&(l_ch&l_ch)";;

LET 'debut' "[m:tri_l_ch]<l_ch,l_ch&l_ch>fst(m)"
      "tri_l_ch->l_ch" ;;

LET 'fin' "[m:tri_l_ch]
      <l_ch,l_ch>snd(<l_ch,l_ch&l_ch>snd(m))"
     "tri_l_ch->l_ch" ;;

LET 'milieu' "[m:tri_l_ch]
       <l_ch,l_ch>fst(<l_ch,l_ch&l_ch>snd(m))"
       "tri_l_ch->l_ch" ;;

LET 'tripl' "[deb:l_ch][mil:l_ch][fini:l_ch]
       <l_ch,l_ch&l_ch>(deb,<l_ch,l_ch>(mil,fini))"
      "l_ch->l_ch->l_ch->tri_l_ch";;
```

Ecriture du programme qui sépare la liste en trois. On utilise des fonctions auxilliaires de type "ch->tri_l_ch->tri_l_ch". Ces fonctions indiquent comment modifier les trois listes suivant la situation.

```
LET 'aj_dec_spa_nil' "[a:ch][p:tri_l_ch]
          (tripl nil_ch (co_ch a (milieu p))(fin p))"
         "ch->tri_l_ch->tri_l_ch";;

LET 'aj_dec_spa_cons' "[a:ch][p:tri_l_ch]
           (tripl nil_ch (co_ch a nil_ch)
           (app_ch (debut p) (app_ch (milieu p) (fin p))))"
          "ch->tri_l_ch->tri_l_ch";;
```

```
LET 'aj_dec_spà "[a:ch][p:tri_l_ch]
                 (consp_dec (debut p)
                 ([h1:(null_ch (debut p))]
                   (aj_dec_spa_nil a p))
                 ([h2:(consp_ch (debut p))]
                   (aj_dec_spa_cons a p)))";;
                 "ch->tri_l_ch->tri_l_ch";;
```

Propriétés de aj_dec_spa en distinguant si la liste du milieu est vide ou non.

```
LET 'co_aj_dec_spa_nil' "[a:ch][p:ch](case1 (null_ch (debut p))
                    (consp_ch (debut p)) (nil_cons_abs (debut p))
                      tri_l_ch (aj_dec_spa_nil a p) (aj_dec_spa_cons a p)
                    (consp_dec (debut p))
                   "[a:ch][p:tri_l_ch]
   (null_ch (debut p))-><tri_l_ch>(aj_dec_spa_nil a p)=(aj_dec_spa a p)";;

LET 'co_aj_dec_spa_cons' "[a:ch][p:ch](case2 (null_ch (debut p))
                    (consp_ch (debut p)) (nil_cons_abs (debut p))
                      tri_l_ch (aj_dec_spa_nil a p) (aj_dec_spa_cons a p)
                    (consp_dec (debut p))
                   "[a:ch][p:tri_l_ch]
  (consp_ch (debut p))-><tri_l_ch>(aj_dec_spa_cons a p)=(aj_dec_spa a p)

LET 'aj_dec_ltr' "[a:ch][p:tri_l_ch]
            (tripl (co_ch a (debut p))(milieu p)(fin p))"
           "ch->tri_l_ch->tri_l_ch";;
```

Le programme de décomposition :

```
let DECOMP = "[l:l_ch](l tri_l_ch
              ([a:ch][p:tri_l_ch]
              (ltr_or_sp a tri_l_ch
              ([h1:(ltr a)](aj_dec_ltr a p))
              ([h2:(sp a)](aj_dec_spa a p))))
              (tripl nil_ch nil_ch nil_ch))";;
LET 'decomp' DECOMP "l_ch->tri_l_ch";;
```

Propriétés de la décomposition :

```
LET 'decomp_ltr' "[a:ch][l:l_ch]
            (case1 (ltr a) (sp a) ([h1:(ltr a)][h2:(sp a)](h2 h1))
            tri_l_ch (aj_dec_ltr a (decomp l))(aj_dec_spa a (decomp l))
            (ltr_or_sp a))";;
           "[a:ch][l:l_ch]
      (ltr a)-><tri_l_ch>(aj_dec_ltr a (decomp l))=(decomp (co_ch a l))";;

LET 'decomp_spà "[a:ch][l:l_ch]
            (case2 (ltr a) (sp a) ([h1:(ltr a)][h2:(sp a)](h2 h1))
            tri_l_ch (aj_dec_ltr a (decomp l))(aj_dec_spa a (decomp l))
            (ltr_or_sp a))";;
           "[a:ch][l:l_ch]
      (ltr a)-><tri_l_ch>(aj_dec_spa a (decomp l))=(decomp (co_ch a l))";;
```

Les parties de la liste prises séparément : Le mot de la fin :

```
LET 'word' "[l:l_ch](debut (decomp l))" "l_ch->l_ch";;
```

Le texte significatif avant le dernier mot :

```
LET 'text' "[l:l_ch](fin (decomp l))" "l_ch->l_ch";;
```

L'espace entre le texte et le dernier mot :

```
LET 'esp' "[l:l_ch](milieu (decomp l))" "l_ch->l_ch";;
```

Le tout rassemblé :

```
LET 'tout' "[l:l_ch](app_ch (word l)(app_ch (esp l) (text l)))" "l_ch->l_ch";;
```

Autres propriétés de la décomposition :

```
LET 'decomp_spa_nil' "[a:ch][l:l_ch][h1:(sp a)][h2:(null_ch (word l))]
                (trans tri_l_ch (aj_dec_spa_nil a (decomp l))
                                (aj_dec_spa a (decomp l))
                                (decomp (co_ch a l))
                (co_aj_dec_spa_nil a (decomp l) h2)
                (decomp_spa a l h1))";;
               "[a:ch][l:l_ch][h1:(sp a)][h2:(null_ch (word l))]
          <tri_l_ch>(aj_dec_spa_nil a (decomp l))=(decomp (co_ch a l))";;

LET 'decomp_spa_cons' "[a:ch][l:l_ch][h1:(sp a)][h2:(consp_ch (word l))]
                (trans tri_l_ch (aj_dec_spa_cons a (decomp l))
                                (aj_dec_spa a (decomp l))
                                (decomp (co_ch a l))
                (co_aj_dec_spa_cons a (decomp l) h2)
                (decomp_spa a l h1))";;
               "[a:ch][l:l_ch][h1:(sp a)][h2:(consp_ch (word l))]
        <tri_l_ch>(aj_dec_spa_cons a (decomp l))=(decomp (co_ch a l))";;
```

Application aux différentes parties de la décomposition :

```
LET 'word_ltr' "[a:ch][l:l_ch][h:(ltr a)]
          (decomp_ltr a l h
          ([p:tri_l_ch]<l_ch>(co_ch a (word l))=(debut p))
          (re l_ch (co_ch a (word l))))"
         "[a:ch][l:l_ch]
          (ltr a)-><l_ch>(co_ch a (word l))=(word (co_ch a l))";;

LET 'esp_ltr' "[a:ch][l:l_ch][h:(ltr a)]
          (decomp_ltr a l h
          ([p:tri_l_ch]<l_ch>(esp l)=(milieu p))
          (re l_ch (esp l)))"
          "[a:ch][l:l_ch](ltr a)-><l_ch>(esp l)=(esp (co_ch a l))";;

LET 'text_ltr' "[a:ch][l:l_ch][h:(ltr a)]
          (decomp_ltr a l h
          ([p:tri_l_ch]<l_ch>(text l)=(fin p))
          (re l_ch (text l)))"
          "[a:ch][l:l_ch](ltr a)-><l_ch>(text l)=(text (co_ch a l))";;
```

```
LET 'word_spa_nil' "[a:ch][l:l_ch][h1:(sp a)][h2:(null_ch (word l))]
                    (decomp_spa_nil a l h1 h2
                    ([p:tri_l_ch]<l_ch>nil_ch=(debut p))
                    (re l_ch nil_ch))"
                    "[a:ch][l:l_ch]
          (sp a)->(null_ch (word l))-><l_ch>nil_ch=(word (co_ch a l))";;

LET 'esp_spa_nil' "[a:ch][l:l_ch][h1:(sp a)][h2:(null_ch (word l))]
                   (decomp_spa_nil a l h1 h2
                   ([p:tri_l_ch]<l_ch>(co_ch a (esp l))=(milieu p))
                   (re l_ch (co_ch a (esp l))))"
                  "[a:ch][l:l_ch]
    (sp a)->(null_ch (word l))-><l_ch>(co_ch a (esp l))=(esp (co_ch a l))";;

LET 'text_spa_nil' "[a:ch][l:l_ch][h1:(sp a)][h2:(null_ch (word l))]
                    (decomp_spa_nil a l h1 h2
                    ([p:tri_l_ch]<l_ch>(text l)=(fin p))
                    (re l_ch (text l)))"
                   "[a:ch][l:l_ch]
          (sp a)->(null_ch (word l))-><l_ch>(text l)=(text (co_ch a l))";;

LET 'word_spa_cons' "[a:ch][l:l_ch][h1:(sp a)][h2:(consp_ch (word l))]
                     (decomp_spa_cons a l h1 h2
                     ([p:tri_l_ch]<l_ch>nil_ch=(debut p))
                     (re l_ch nil_ch))"
                    "[a:ch][l:l_ch]
       (sp a)->(consp_ch (word l))-><l_ch>nil_ch=(word (co_ch a l))";;

LET 'esp_spa_cons' "[a:ch][l:l_ch][h1:(sp a)][h2:(consp_ch (word l))]
                    (decomp_spa_cons a l h1 h2
                    ([p:tri_l_ch]<l_ch>(co_ch a nil_ch)=(milieu p))
                    (re l_ch (co_ch a nil_ch)))"
                   "[a:ch][l:l_ch]
    (sp a)->(consp_ch (word l))-><l_ch>(co_ch a nil_ch)=(esp (co_ch a l))";;

LET 'text_spa_cons' "[a:ch][l:l_ch][h1:(sp a)][h2:(consp_ch (word l))]
                     (decomp_spa_cons a l h1 h2
                     ([p:tri_l_ch]<l_ch>(tout l)=(fin p))
                     (re l_ch (text l)))"
                    "[a:ch][l:l_ch]
          (sp a)->(consp_ch (word l))-><l_ch>(tout l)=(text (co_ch a l))";;

LET 'word_spà "[a:ch][l:l_ch][h:(sp a)]
              (consp_dec (word l) <l_ch>nil_ch=(word(co_ch a l))
              ([h1:(null_ch (word l))]
              (word_spa_nil a l h h1))
              ([h2:(consp_ch (word l))]
              (word_spa_cons a l h h1)))"
             "[a:ch][l:l_ch](sp a)->(<l_ch>nil_ch=(word(co_ch a l)))";;
```

On appelle "rest" tout ce qui n'est pas le premier mot :

```
LET 'rest' "[l:l_ch](app_ch (esp l) (text l))" "l_ch->l_ch";;
```

On prouve que ce rest vérifie la propriété end_void :

```
PROP 'esp_void_prop' "[l:l_ch](end_void (rest l))";;

let ESP_VOID = "[l:l_ch](pred_ch l esp_void_prop
                end_void_nil
                ([a:ch][m:l_ch][h:(esp_void_prop m)]
                (ltr_or_sp a (esp_void_prop (co_ch a m))
                ([h1:(ltr a)](esp_ltr a m h1
                ([m':l_ch](end_void (app_ch m' (text (co_ch a m)))))
                (text_ltr a m h1 ([m':l_ch](end_void (esp m) m')) h)))

                ([h2:(sp a)]
                (consp_dec (word l) (esp_void_prop (co_ch a m))
                ([h21:(null_ch (word l))]
                (text_spa_nil a m h2 h21
                ([m':l_ch](end_void (app_ch m' (text (co_ch a m)))))
                (end_void_co a
                (app_ch (esp m) (text (co_ch a m))) h2)))

                ([h22:(consp_ch (word l))]
                (text_spa_cons a m h2 h22
                ([m':l_ch](end_void (app_ch m' (text (co_ch a m)))))
                (end_spa_co a
                (app_ch nil_ch (text (co_ch a m))) h2))))))))";;
LET 'esp_void' "[l:l_ch](end_void (rest l))";;
```

En recomposant les trois morceaux, on obtient la liste initiale :

```
let L_TOUT_L = "[l:l_ch](pred_ch l ([m:l_ch] <l_ch>m=(tout m))
        (re l_ch nil_ch)
        ([a:ch][m:l_ch][h:<l_ch>m=(tout m)]
        (ltr_or_sp a (<l_ch>(co_ch a m)=(tout (co_ch a m)))

        ([h1:(ltr a)]
        (word_ltr a m h1
        ([m1:l_ch]
    <l_ch>(co_ch a m)=(app_ch m1
            (app_ch (esp (co_ch a m))(text (co_ch a m)))))
        (esp_ltr a m h1
        ([m1:l_ch]
    <l_ch>(co_ch a m)=(app_ch (co_ch a (text m))
            (app_ch m1 (text (co_ch a m)))))
        (text_ltr a m h1
        ([m1:l_ch]
    <l_ch>(co_ch a m)=(app_ch (co_ch a (text m))(app_ch (esp m) m1)))
        (h ([m1:l_ch]<l_ch>(co_ch a m)=(co_ch a m1))
        (re_l_ch (co_ch a m)))))))
```

```
        ([h2:(sp a)](consp_dec (word m)
        (<l_ch>(co_ch a m)=(tout (co_ch a m)))
        ([h21:(null_ch (word m))]
        (word_spa_nil a m h2 h21
        ([m1:l_ch]
    <l_ch>(co_ch a m)=(app_ch m1
                (app_ch (esp (co_ch a m))(text (co_ch a m)))))
        (esp_spa_nil a m h2 h21
        ([m1:l_ch]
         <l_ch>(co_ch a m)=(app_ch m1 (text (co_ch a m))))
        (text_spa_nil a m h2 h21
        ([m1:l_ch]
         <l_ch>(co_ch a m)=(app_ch (co_ch a (esp m)) m1))
        (h ([m1:l_ch]<l_ch>(co_ch a m)=(co_ch a m1))
        (re_l_ch (co_ch a m)))))))

         ([h22:(consp_ch (word m))]
        (word_spa_cons a m h2 h22
        ([m1:l_ch]
    <l_ch>(co_ch a m)=(app_ch m1
                (app_ch (esp (co_ch a m))(text (co_ch a m)))))
        (esp_spa_cons a m h2 h22
        ([m1:l_ch]
         <l_ch>(co_ch a m)=(app_ch m1 (text (co_ch a m))))
        (text_spa_cons a m h2 h22
        ([m1:l_ch]
    <l_ch>(co_ch a m)=(app_ch (co_ch a (nil_ch)) m1))
        (h ([m1:l_ch]<l_ch>(co_ch a m)=(co_ch a m1))
        (re_l_ch (co_ch a m))))))))))))";;
```

LET 'l_tout_l' L_TOUT_L "[l:l_ch]<l_ch>l=(tout l)";;

On montre aussi que le reste est équivalent à la troisième partie normalisée à laquelle on a ajouté un lf.

```
PROP 'esp_lf_prop' "[l:l_ch]
            (equiv (rest l)(co_ch lf (text l)))";;

let ESP_LF = "[l:l_ch](pred_ch l esp_lf_prop
        (norm_sp_void lf nil_ch end_void_nil sp_lf)
        ([a:ch][l':l_ch][h:(esp_lf_prop l')]
        (ltr_or_sp a (esp_lf_prop (co_ch a l'))

        ([h1:(ltr a)]
        (esp_ltr a l' h1
         ([m:l_ch](equiv (app_ch m (text (co_ch a l')))
                  (co_ch lf (text (co_ch a l')))))
        (text_ltr a l' h1
        ([m:l_ch](equiv (app_ch (esp l') m) (co_ch lf m))) h)))
```

```
([h2:(sp a)]
(consp_dec (word l') (esp_lf_prop (co_ch a l'))
([h21:(null_ch (word l))]
(esp_spa_nil a l' h2 h21
([m:l_ch](equiv (app_ch m (text (co_ch a l')))
          (co_ch lf (text (co_ch a l')))))
(text_spa_nil a l' h2 h21
([m:l_ch](equiv (app_ch (co_ch a (esp l')) m) (co_ch lf m)))

(tran (norm (app_ch (co_ch a (esp l'))(text l')))
    (norm (app_ch (esp l') (text l')))
    (norm (co_ch lf (text l')))
    (sym l_ch
    (norm (app_ch (esp l') (text l')))
    (norm (co_ch a (app_ch (esp l')(text l'))))
    (norm_sp_void a (app_ch (esp l') (text l'))
    h2 (esp_void l'))) h ))))

([h22:(consp_ch (norm l'))]
(esp_spa_cons a l' h2 h22
 ([m:l_ch](equiv (app_ch m (text (co_ch a l')))
          (co_ch lf (text (co_ch a l')))))
(eq_spa a (text (co_ch a l')) lf (text (co_ch a l'))
(eq_re (text (co_ch a l'))) h2 sp_lf))))))))";;
```

```
LET 'esp_lf' ESP_LF "[l:l_ch]
          (equiv (rest l) (co_ch lf (text l)))";;
```

AUTOUR DE LA PROPRIETE ETRE FORMATTE

On introduit une condition pour que la mise en page soit possible : les mots ne doivent pas être de longueur supérieure à la longueur maximale.

```
AXIOM 'LE_wd_max' "[l:l_ch](LE (lgth_ch (word l)) max)";;
```

Voici maintenant la définition des prédicats sur la longueur de la dernière ligne du texte : cette liste est pleine ou non :

```
PROP 'line_full' "[l:l_ch]<Nat>max=(lgth_ch (buffer l))";;

PROP 'line_no_full' "[l:l_ch](LT (lgth_ch (buffer l)) max)";;
```

Définition du prédicat "être formatté" par cas.

Si la dernière ligne n'est pas pleine on peut mettre un blanc après une lettre.

```
PROP 'cas1' "{P|l_ch->*}[l:l_ch]
        [h1:(line_no_full l)]
        [h2:(end_ltr l)]
        (P l)->(P (co_ch bl l))";;
```

On peut toujours mettre une lettre sur une ligne non pleine.

```
PROP 'cas2' "{P|l_ch->*}[l:l_ch][a:ch]
        [h1:(line_no_full l)][h2:(ltr a)]
        (P l)->(P (co_ch a l))";;
```

Si la ligne est pleine, on déplace le dernier mot pour insérer une lettre.

```
LET 'coup' "[l:l_ch]
        (app_ch (word l)(co_ch lf (text l)))" ;;
       "ch->l_ch->l_ch";;

PROP 'cas3' "{P|l_ch->*}[l:l_ch][a:ch]
        [h1:(line_full l)][h2:(ltr a)](P l)->(P (co_ch a (coup l)))";;
```

Si la ligne est pleine on peut inserer un lf apres une lettre

```
PROP 'cas4' "{P|l_ch->*}[l:l_ch][h1:(line_full l)][h2:(end_ltr l)]
         (P l)->(P (co_ch lf l))";;
```

Définition de "format" :

```
PROP 'format' "[l:l_ch]{P|l_ch->*}[h1:(P nil_ch)]
          (cas1 P)->(cas2 P)->(cas3 P)->(cas4 P)->(P l)";;
```

SPECIFICATION DU PROGRAMME

```
PROP 'form_spec' "[l:l_ch]<l_ch>Sig([m:l_ch](equiv l m)&(format m))";;
```

Abréviations utiles :

```
PROP 'form_prop' "[l:l_ch][m:l_ch](equiv l m)&(format m)";;

LET 'form_pair' "[l:l_ch][m:l_ch][h1:(equiv l m)][h2:(format m)]
          (exist l_ch (form_prop l) m
          <(equiv l m),(format m)>(h1,h2))"
         "[l:l_ch][m:l_ch]
          (equiv l m)->(format m)->(form_spec l)";;
```

PROPRIETES DU BUFFER

On montre que le buffer est toujours de longueur inférieure à max :

```
PROP 'buff_post_lf_prop' "[l:l_ch][m_l_ch]
            <l_ch>(buffer l)=(buffer (app_ch l (co_ch lf m)))";;

let BUFF_POST_LF = "[l:l_ch][m:l_ch](pred_ch l buff_post_lf_prop
            (buffer_lf lf m (re ch lf))
            ([a:ch][n:l_ch][h:(buff_post_lf_prop n)]
            (decid_lf a (buff_post_lf_prop (co_ch a n))

            ([h1:~<ch>lf=a]
            (buffer_no_lf a n h1
        ([n':l_ch]<l_ch>n'=(buffer (app_ch (co_ch a n) (co_ch lf m))))
            (buffer_no_lf a (app_ch n (co_ch lf m)) h1
        ([n':l_ch]<l_ch>(co_ch a (buffer n))=n')
            (h ([n':l_ch]<l_ch>(co_ch a (buffer n))=(co_ch a n'))
            (re ch (co_ch a (buffer n)))))))
```

```
                ([h2:<ch>lf=a]
                (buffer_lf a (app_ch n (co_ch lf m)) h2
                ([n':l_ch]<l_ch>(buffer (co_ch a n))=n')
                (buffer_lf a l h2
                ([n':l_ch]<l_ch>n'=nil_ch)
                (re l_ch nil_ch)))))))";;

LET 'buff_post_lf' BUFF_POST_LF "[l:l_ch][m:l_ch]
                <l_ch>(buffer l)=(buffer (app_ch l (co_ch lf m)))";;

PROP 'buff_word_prop' "[l:l_ch]<l_ch>(word l)=(buffer (word l))";;

let BUFFER_WORD = "[l_l_ch](pre_ch l buff_word_prop
            (re l_ch nil_ch)
            ([a:ch][m:l_ch][h:(buff_word_prop m)]
            (ltr_or_sp a (buff_word_prop (co_ch a m))
            ([h1:(ltr a)](word_ltr a m h1
            ([m':l_ch]<l_ch>m'=(buffer m'))
            (buffer_no_lf a (word m) (ltr_no_lf a h1)
            ([m':l_ch]<l_ch>(co_ch a (word m))=m')
            (h ([m':l_ch](co_ch a (word m))=(co_ch a m'))
            (re l_ch (co_ch a (word m)))))))
            ([h2:(sp a)](word_spa a m h2
            ([m':l_ch]<l_ch>m'=(buffer m'))(re l_ch nil_ch))))))";;
LET 'buffer_word' BUFFER_WORD "[l:l_ch]<l_ch>(word l)=(buffer (word l))";

let BUFFER_COUP = "[l:l_ch](tran l_ch (word l) (buffer (word l))
                    (buffer (coup l))
                    (buffer_word l)
                    (buffer_post_lf (word l) (text l)))";;
LET 'buffer_coup' BUFFER_COUP "[l:l_ch]<l_ch>(word l)=(buffer (coup l))";;

let POSSIBLE = "[l:l_ch][h1:(format l)]
            (h1 ([m:l_ch](LE (lgth_ch (buffer m)) max))
            (peano max)
            ([m:l_ch][g1:(line_no_full m)]
            [g2:(end_ltr m)][g3:(LE (lgth_ch (buffer m)) max)]
            (buffer_no_lf bl m (bl_no_lf bl (re ch bl))
            ([m':l_ch](LE (lgth_ch m') max)) g1))
            ([m:l_ch][a:ch][g1:(line_no_full m)][g2:(ltr a)]

            ([g3:(LE (lgth_ch (buffer m)) max)]
            (buffer_no_lf a m (ltr_no_lf a h2)
            ([m':l_ch](LE (lgth_ch m') max)) g1)))
            ([m:l_ch][a:ch][g1:(line_full m)][g2:(ltr a)]
            [g3:(LE (lgth_ch (buffer m)) max)]

            (buffer_coup (co_ch a m)
            ([m':l_ch](LE (lgth_ch m') max))
            (LE_wd_max (co_ch a m))))
            ([m:l_ch][g1:(line_full m)][g2:(end_ltr m)]
            [g3:(LE (lgth_ch (buffer m)) max)]
            (buffer_lf lf m (re ch lf)
            ([m':l_ch](LE (lgth_ch m') max))(peano max))))";;
```

```
LET 'possiblè POSSIBLE "[l:l_ch](format l)->(LE (lgth_ch (buffer l)) max)";;
```

DEMONSTRATION DU PROGRAMME :

Cas de la liste vide

```
LET 'format_nil' "{P|l_ch->*}[h1:(P nil_ch)][h2:(cas1 P)][h3:(cas2 P)]
                  [h4:(cas3 P)][h5:(cas4 P)]h1"
                 "(format nil_ch)";;

let FORM_NIL = "(form_pair nil_ch nil_ch (eq_re nil_ch) format_nil)";;
LET 'form_nil' FORM_NIL "(form_spec nil_ch)";;
```

Pas de récurrence On envisage les différents cas :

```
let LEM1 = "[a:ch][m:l_ch][h1:(format m)][h2:(ltr a)]
           [h3:(line_no_full m)]
           {P|l_ch->*}[h4:(P nil_ch)][h5:(cas1 P)][h6:(cas2 P)]
           [h7:(cas3 P)][h8:(cas4 P)](h6 m a h3 h2 (h1 P h4 h5 h6 h7 h8))";;
LET 'lem1' LEM1 "[a:ch][m:l_ch]
          (format m)->(ltr a)->(line_no_full m)->(format (co_ch a m))";;

let CONS1 = "[a:ch][l:l_ch][m:l_ch][h1:(equiv l m)][h2:(format m)]
            [h3:(line_no_full l)][h4:(ltr a)]
            (form_pair (co_ch a l) (co_ch a m)
            (eq_ltr a l m h1 h4) (lem1 a m h2 h4 h3))";;
LET 'cons1' CONS1 "[a:ch][l:l_ch][m:l_ch][h1:(equiv l m)][h2:(format m)]
                   (line_no_full l)->(ltr a)->(form_spec (co_ch a l))";;

let LEM2 = "[m:l_ch][h1:(format m)]
           [h2:(line_no_full m)][h3:(end_ltr m)]
           {P|l_ch->*}[h4:(P nil_ch)][h5:(cas1 P)][h6:(cas2 P)]
           [h7:(cas3 P)][h8:(cas4 P)](h5 m h2 h3 (h1 P h4 h5 h6 h7 h8))";;
LET 'lem2' LEM2 "[m:l_ch]
  (format m)->(line_no_full m)->(end_ltr m)->(format (co_ch bl m))";;

let CONS2 = "[a:ch][l:l_ch][m:l_ch][h1:(equiv l m)][h2:(format m)]
            [h3:(line_no_full m)][h4:(sp a)][h5:(end_ltr m)]
            (form_pair (co_ch a l) (co_ch bl m)
            (eq_sp a l b m h1 h4 sp_bl)(lem2 m h2 h3 h5))";;
LET 'cons2' CONS2 "[a:ch][l:l_ch][m:l_ch][h1:(equiv l m)][h2:(format m)]
          (line_no_full m)->(sp a)->(end_ltr m)->(form_spec (co_ch a l))";;

let LEM3 = "[a:ch][m:l_ch][h1:(format m)][h2:(line_full m)][h3:(ltr a)]
           {P|l_ch->*}[h4:(P nil_ch)][h5:(cas1 P)][h6:(cas2 P)]
           [h7:(cas3 P)][h8:(cas4 P)](h7 m a h2 h3 (h1 P h4 h5 h6 h7 h8))";;
LET 'lem3' LEM3 "[a:ch][m:l_ch]
           (format m)->(line_full m)->(ltr a)->(format (co_ch a (coup m)))";;

LET 'eq_coup' "[l:l_ch](sym l (tout l) (l_tout_l l)
               ([m:l_ch] (equiv m (coup l)))
               (eq_app (rest l) (co_ch lf (text l)) (eq_lf l) (word l)))"
              "[l:l_ch](equiv l (coup l))";;
```

```
let CONS3 = "[a:ch][l:l_ch][m:l_ch][h1:(equiv l m)][h2:(format m)]
        [h3:(line_full m)][h4:(ltr a)]
        (form_pair (co_ch a l) (co_ch a (coup l))
        (eq_co a l (coup m) (eq_tran l m (coup m) h1 (eq_coup m)))
        (lem3 a m h2 h3 h4))";;
LET 'cons3' CONS3 "[a:ch][l:l_ch][m:l_ch]
(equiv l m)->(format m)->(line_full m)->(ltr a)->(form_spec (co_ch a l))";;

let LEM4 = "[m:l_ch][h1:(format m)][h2:(line_full m)][h3:(end_ltr m)]
       {P|l_ch->*}[h4:(P nil_ch)][h5:(cas1 P)][h6:(cas2 P)]
       [h7:(cas3 P)][h8:(cas4 P)](h8 m h2 h3 (h1 P h4 h5 h6 h7 h8))";;
LET 'lem4' LEM4 "[m:l_ch]
       (format m)->(line_full m)->(end_ltr m)->(format (co_ch lf m))";;

let CONS4 = "[a:ch][l:l_ch][m:l_ch][h0:(sp a)][h1:(equiv l m)][h2:(format m)]
        [h3:(line_full m)][h4:(end_ltr m)]
        (form_pair (co_ch a l) (co_ch lf m)
        (eq_spa a l lf m h1 h0 sp_lf)
        (lem4 m h2 h3 h4))";;
LET 'cons4' CONS4 "[a:ch][l:l_ch][m:l_ch][h0:(sp a)][h1:(equiv l m)]
      (format m)->(line_full m)->(end_ltr m)->(form_spec (co_ch a l))";;

let CONS5 = "[a:ch][l:l_ch][m:l_ch][h0:(sp a)][h1:(equiv l m)]
        [h2:(format m)][h3:(end_void m)]
        (form_pair (co_ch a l) m
        (tran_eq (co_ch a l) (co_ch a m) m
        (eq_co a l m h1)
        (sym l_ch (norm m) (norm (co_ch a m))
        (norm_sp_void a m  h3 h0))) h2)";;
LET 'cons5' CONS5 "[a:ch][l:l_ch][m:l_ch]
   (sp a)->(equiv l m)->(format m)->(end_void m)->(form_spec (co_ch a l))";

let FORM_CONS = "[a:ch][l:l_ch][m:l_ch][h1:(equiv l m)][h2:(format m)]
           (LE_EG_LT (lgth_ch (buffer m)) max (possible m h2)
           (form_spec (co_ch a l))
           ([g1:(line_full m)](ltr_or_sp a (form_spec (co_ch a l))
           ([g11:(ltr a)](cons3 a l m h1 h2 g1 g11))
           ([g12:(sp a)](end_void_ltr m (form_spec (co_ch a l))
           ([g121:(end_void m)](cons5 a l m g12 h1 h2 g121))
           ([g122:(end_ltr m)](cons4 a l m g12 h1 h2 g1 g122))))))
           ([g2:(line_no_full m)](ltr_or_sp a (form_spec (co_ch a l))
           ([g21:(ltr a)](cons1 a l m h1 h2 g2 g21))
           ([g22:(sp a)](end_void_ltr m (form_spec (co_ch a l))
           ([g221:(end_void m)](cons5 a l m g22 h1 h2  g221))
           ([g222:(end_ltr m)](cons2 a l m h1 h2 g2 g22 g222)))))))";;
LET 'form_cons' FORM_CONS "[a:ch][l:l_ch][m:l_ch]
                (equiv l m)->(format m)->(form_spec (co_ch a l))";;
```

PROGRAMME FINAL

```
let FORM_PROG = "[l:l_ch](pred_ch l form_spec form_nil
                  ([a:ch][m:l_ch][h:(form_spec l)]
                  (h (form_spec co_ch a m)
                  ([m:l_ch][g:(form_prop m)]
                  (g (form_spec (co_ch a m)) (form_cons a l m))))))";;
LET 'form_prog' FORM_PROG 'form_spec';;
```

4.3. Remarques

La principale remarque sur cette étude est de noter l'utilisation de la quantification sur les prédicats pour obtenir des définitions telles que celles de ***format***, ***end_void*** ou ***equiv***. On utilise là encore la facilité d'écrire dans le calcul des constructions des petits programmes tels que la recherche de la dernière ligne. Bien sûr, on peut reprocher à ce programme sa longueur et le nombre important de preuves de petites trivialités. En fait la difficulté est purement syntaxique, il n'y a qu'à se laisser guider par les types pour écrire les termes.

Une dernière remarque sur la définition de ***format***. Celle-ci peut être contestée. Mais quelle que soit la définition que l'on donne de cette propriété pour montrer que le programme présenté ici effectue bien une mise en page, il suffira de montrer (éventuellement) à la main) que la nouvelle propriété est stable par les constructeurs de ***format***. Si on appelle *Formatbis* une autre définition il suffit de montrer :

```
1_ (Formatbis (nil ch))
2_ ([a:ch][m:(l_ch)]
     (line_no_full l)->(ltr a)->
     (Formatbis l)->(Formatbis (co_ch a l)))
3_ ([m:(l_ch)]
     (line_no_full l)->(end_ltr l)->
     (Formatbis l)->(Formatbis (co_ch bl l)))
4_ ([m:(l_ch)]
     (line_full l)->(end_ltr l)->
     (Formatbis l)->(Formatbis (co_ch lf l))
5_ ([a:ch][m:(l_ch)]
     (line_full l)->(ltr a)->
     (Formatbis l)->(Formatbis (co_ch a (coup l))))
```

5. Quicksort

5.1. Présentation

Rappelons l'algorithme :

> On a n éléments à trier, si n = 0 c'est évident sinon on choisit un élement a, on sépare en deux groupes les n-1 élements restant suivant s'ils sont plus grand que a ou non, on applique récursivement le procédé aux deux groupes puis on réunit les morceaux.

Ici nous n'utiliserons pas des tableaux mais des listes, en particulier on ne s'intéresse pas à l'astucieuse étape de décomposition de Quicksort qui permet de ne déplacer que peu d'objets. On ne peut pas utiliser, comme pour l'algorithme de mise en page, le principe de récurrence linéaire sur les listes puisque l'on fait un double appel à des listes dont on saura juste qu'elles sont de longueur inférieure. Le principe à utiliser est ici une une récurrence noethérienne générale sur la longueur des listes que nous commencons par prouver en utilisant une récurrence sur les entiers.

La spécification générale du programme est :

$$\forall\, l \in (listA)\ \exists\, m \in (listA)\cdot(tri\ m)\&(permut\ l\ m)$$

Encore faut-il définir *tri*, c'est-à-dire ce qu'est une liste triée et *permut* c'est-à-dire quand deux listes sont égales à permutation près. Pour cela on commence par définir *inf_list* et *sup_list*, (*inf_list a l*) (resp (*sup_list a l*)) signifie que a est inférieur (resp supérieure) aux éléments de *l*.

On commence par écrire un petit programme pour décomposer une liste en fonction de la position relative de ces éléments par rapport à un élément donné. La spécification de ce programme est :

$$\forall\, a \in A, \forall\, l \in (listA)\ \exists\, m_1, m_2$$

$$sup\_list\ a\ m_1\ \&\ inf\_list\ a\ m_2\ \&\ permut\ l\ (append\ A\ m_1\ m_2)$$

Le corps du programme consiste à montrer que la spécification du programme complet de tri vérifie la condition de stabilité noethérienne, c'est-à-dire à montrer comment on construit une liste triée equivalente à *l* quand on sait le faire pour toutes les listes de longueur strictement inférieure. Le cas où *l=nil* est trivial sinon *l=a.m* et on sépare *m* en deux par rapport à *a*. Il nous reste alors à montrer que les propriétés souhaitées sont bien respectées.

5.2. Programme

Définition d'une relation binaire totale et d'un préordre :

```
let TOTAL = "!A {R|A->A->*}[a:A][b:A](R a b)+(R b a)";;
PROP 'total' TOTAL;;

PROP 'pre_ordrè "!A.{R|A->A->*}(refl A R)&(trans A R)";;
```

On suppose que l'on dispose d'un ensemble A et d'un préordre total inf sur A :

```
DECL 'A' "*";;
DECL 'inf' "A->A->*";;
AXIOM 'ord' "(pre_ordre A inf)";;
AXIOM 'tot' "(total A inf)";;
```

On utilisera la réflexivité de inf :

```
LET 'refl_inf' "<(refl A inf),(trans A inf)>fst(ord)"
        "[a:A](inf a a)";;
```

On particularise les constructions sur les listes aux listes d'éléments de A.

Fonctions de base :

```
PROP 'List' "(list A)";;
LET 'Cons' "(cons A)" "A->List->List";;
LET 'Append' "(append A)" "List->List->List";;

LET 'Mil' "[m1:List][a:A][m2:List](Append m1 (Cons a m2))"
      "List->A->List->List";;
LET 'Nil' "(nil A)" "List";;
LET 'Lgth' "(length A)" "List->Nat";;
```

L 'axiome de construction des listes dont le type est :
[l:List]{P|List->}(P Nil)->([a:A][m:List](P m)->(P (Cons a m)))->(P l)*

```
LET 'PList' "(axi_list A)" "(pred_list A)";;
```

Une liste est soit vide, soit le cons d'un élément et d'une liste.

```
PROP 'Nullp' "(nullp A)";;
PROP 'Consp' "(consp A)";;
LET 'Listp' "(case_list A)" "(listp A)";;
```

L'associativité de Append

```
LET 'Assoc_app' "(assoc_app A)"
          "[l:List][m:List][n:List]
          <List>(Append l (Append m n))=(Append (Append l m) n)";;
```

La longueur de (Append m n) est la somme des longueurs de m et n :

```
LET 'Lgth_App' "(lgth_app A)" "[l:List][m:List]
                <Nat>(add (Lgth l)(Lgth m))=(Lgth (Append l m))";;
```

Démonstration de l'induction noethérienne sur la longueur des listes. On veut montrer que : soit P une propriété sur les listes, si on sait déduire P pour une liste l dès que l'on sait que P est vrai pour toutes les listes de longueur strictement inférieure (cette propriété est nommée noeth_ind) alors on sait que P est vrai pour toutes les listes. On a ainsi un schéma d'induction permettant de prouver la terminaison des programmes récurrents où la longueur des listes décroit lors des appels. On a introduit l'axiome axi_ari de type (S x)=0 —> {} on en déduit aisément la proprieté x<0 —> {} :

```
let ABS_ARI = "[n:Nat][h:(LT n 0)]
        (h ([x:Nat]~(<Nat>x=0)) (axi_ari n)
          ([x:Nat][h:~(<Nat>x=0)](axi_ari x)) (re Nat 0))";;
LET 'abs_ari' ABS_ARI "[n:Nat](LT n 0)->{}";;
```

Ordre sur les longueurs de listes :

```
PROP 'LT_Lgth' "[l:List][m:List](LT (Lgth l) (Lgth m))";;
```

Propriété pour cet ordre d'être noethérien :

```
PROP 'noeth_ind' "{P|List->*}[l:List]([m:List](LT_Lgth m l)->(P m))->(P l)";;
```

La récurrence noethérienne sur les listes se ramène à une récurrence sur les entiers.

```
PROP 'Nat_noeth_prop' "{P|List->*}[n:Nat][l:List](LT (Lgth l) n)->(P l)";;

let NAT_NOETH = "{P|List->*}[h1:(noeth_ind P)][n:Nat]
          (peano n (Nat_noeth_prop P)
          ([l:List][h2:(LT (Lgth l) 0)]
          (abs_ari (Lgth l) h2 (P l)))
          ([m:Nat][h3:(Nat_noeth_prop P m)][l:List]
          [h4:(LT (Lgth l) (S m))]
          (LE_EGLT (Lgth l) m (S_LE (Lgth l) m h4) (P l)
          ([t1:<Nat>(Lgth l)=m](h1 l ([l':List][h5:(LT_Lgth l' l)]
          (h3 l' (t1 ([m':Nat](LT (Lgth l') m')) h5)))))
          ([t2:(LT (Lgth l) m)](h3 l t2)))))";;
LET 'Nat_noeth' NAT_NOETH "{P|List->*}(noeth_ind P)->(Nat_noeth_prop P)";;

LET 'List_noeth' "[l:List]{P|List->*}[h:(noeth_ind P)]
          (Nat_noeth P h (S (Lgth l)) l (LE_n_n (S (Lgth l))))"
          "[l:List]{P|List->*}(noeth_ind P)->(P l)";;
```

Prédicats sur les listes :

On se donne rel un prédicat sur A et on définit rel_list comme étant la propriété pour une liste d'être composée d'éléments qui vérifient rel :

```
let REL_LIST = "{rel|A->*}[l:List]{P|List->*}
        (P Nil)->([b:A][m:List](rel b)->(P m)->(P (Cons b m)))->(P l)";;
PROP 'rel_list' REL_LIST;;
```

Cas particuliers : les listes d'éléments plus petits ou plus grands qu'un certain objet :

```
PROP 'inf_list' "[a:A](rel_list (inf a))";;

PROP 'sup' "[a:A][b:A](inf b a)";;

PROP 'sup_list' "[a:A](rel_list (sup a))";;
```

Définition constructive du prédicat être triée : la liste vide est triée et si m1 m2 sont deux listes triées, a plus grand que tous les éléments de m1 et plus petit que tous les élements de m2 alors la liste append m1 a.m2 est triée.

```
PROP 'ins_mil' "{P|List->*}[m1:List][m2:List][a:A]
   (P m1)->(P m2)->(sup_list a m1)->(inf_list a m2)->(P (Mil m1 a m2))";;

PROP 'tri' "[l:List]{P|List->*}(P Nil)->(ins_mil P)->(P l)";;
```

Propriétés évidentes de tri :

```
LET 'tri_nil' "{P|List->*}[h1:(P Nil)][h2:(ins_mil P)]h1" "(tri Nil)";;
```

```
let TRI_APP = "[l1:List][l2:List][a:A][h1:(tri l1)][h2:(tri l2)]
        [h3:(sup_list a l1)][h4:(inf_list a l2)]
        {P|List->*}[h5:(P Nil)][h6:(ins_mil P)]
        (h6 l1 l2 a (h1 P h5 h6) (h2 P h5 h6) h3 h4)";;
LET 'tri_app' TRI_APP "[l1:List][l2:List][a:A]
 (tri l1)->(tri l2)->(sup_list a l1)->(inf_list a l2)->(tri (Mil l1 a l2))";;
```

Définition de la relation sur les listes : être égales à permutation près.

```
PROP 'ins_app_mil' "{P|List->List->*}[a:A][l:List][m1:List][m2:List]
        (P l (Append m1 m2))->(P (Cons a l) (Mil m1 a m2))";;

PROP 'permut' "[l:List][m:List]{P|List->List->*}
     (P Nil Nil)->(ins_app_mil P)->(trans List P)->(P l m)";;
```

Propriétés évidentes. Il y a trois moyens d'obtenir des listes qui permutent. Avoir deux listes vides, un argument de transitivité ou bien par l'ajout d'un même élément en tête et au milieu de deux listes qui permutent.

```
LET 'perm_nil' "{P|List->List->*}[h1:(P Nil Nil)][h2:(ins_app_mil P)]
        [h3:(trans List P)]h1"
       "(permut Nil Nil)";;

LET 'perm_tran' "[l:List][m:List][n:List][c1:(permut l m)][c2:(permut m n)]
        {P|List->List->*}[h1:(P Nil Nil)][h2:(ins_app_mil P)]
        [h3:(trans List P)](h3 l m n (c1 P h1 h2 h3)(c2 P h1 h2 h3))"
       "(trans List permut)";;

LET 'perm_cons' "[a:A][l:List][m1:List][m2:List][c:(permut l (Append m1 m2))]
        {P|List->List->*}[h1:(P Nil Nil)][h2:(ins_app_mil P)]
        [h3:(trans List P)](h2 a l m1 m2 (c P h1 h2 h3))"
       "[a:A][l:List][m1:List][m2:List]
      (permut l (Append m1 m2))->(permut (Cons a l) (Mil m1 a m2))";;
```

Spécification d'un programme de tri :

```
PROP 'sort' "[l:List][m:List](tri m)&(permut l m)";;

PROP 'spec_tri' "[l:List]<List>Sig((sort l))";;
```

Un constructeur bien utile :

```
LET 'sort_ex' "[l:List][m:List][h1:(tri m)][h2:(permut l m)]
        (exist List (sort l) m <(tri m),(permut l m)>(h1,h2))"
       "[l:List][m:List](tri m)->(permut l m)->(spec_tri l)";;
```

Il n'est pas difficile de trier la liste vide.

```
LET 'sort_nil' "(sort_ex Nil Nil tri_nil perm_nil)" "(spec_tri Nil)";;
```

Nous voulons montrer des propriété de stabilité sur les listes vérifiant un prédicat. On montre d'abord les propriétés évidentes liées a la structure de la définition (rel_list_nil et rel_list_cons) puis nous prouvons que si l et m permutent et si l vérifie rel alors m le verifie aussi.

```
PROP 'rel_rec' "{rel|A->*}{P|List->*}[b:A][l:List]
        (rel b)->(P l)->(P (Cons b l))";;
```

```
LET 'rel_list_nil' "{rel|A->*}{P|List->*}
                    [h1:(P Nil)][h2:(rel_rec rel P)]h1"
                   "{rel|A->*}(rel_list rel Nil)";;

LET 'rel_list_cons' "{rel|A->*}[b:A][l:List]
                     [h:(rel b)][h':(rel_list rel l)]
                     {P|List->*}[h1:(P Nil)][h2:(rel_rec rel P)]
                     (h2 b l h (h' P h1 h2))"
                    "{rel|A->*}[b:A][l:List]
                     (rel b)->(rel_list rel l)->(rel_list rel (Cons b l))";;
```

On applique ces résultats aux prédicats inf et sup :

```
LET 'inf_list_nil' "[a:A](rel_list_nil (inf a))" "[a:A](inf_list a Nil)";;

LET 'sup_list_nil' "[a:A](rel_list_nil (sup a))" "[a:A](sup_list a Nil)";;

LET 'inf_list_cons' "[a:A](rel_list_cons (inf a))"
                    "[a:A][b:A][l:List]
                     (inf a b)->(inf_list a l)->(inf_list a (Cons b l))";;

LET 'sup_list_cons' "[a:A](rel_list_cons (sup a))"
                    "[a:A][b:A][l:List]
                     (inf b a)->(sup_list a l)->(sup_list a (Cons b l))";;
```

On montre que si la liste b.l vérifie rel alors b et l vérifient rel l'utilisation de la fonction hd_tot nous oblige à supposer l'existence d'un élément a qui vérifie l.

```
LET 'rel_hd' "{rel|A->*}[a:A][h0:(rel a)]
              [b:A][l:List][h:(rel_list rel (Cons b l))]
              (h ([m:List](rel (hd_tot A a m))) h0
              ([c:A][m':List][h':(rel c)][h'':(rel (hd_tot A a m'))] h'))"
             "{rel|A->*}[a:A][h0:(rel a)][b:A][l:List]
              (rel_list rel (Cons b l))->(rel b)";;
```

Pour montrer que l verifie rel on montre en fait par récurrence que la liste l et cette liste sans son premier élément vérifient rel.

```
PROP 'rel_tl_prop' "{rel|A->*}[l:List]
                    (rel_list rel (tl_tot A l))";;

PROP 'rel_aux' "{rel|A->*}[l:List]
                (rel_tl_prop rel l)&(rel_list rel l)";;
```

Quelques primitives pour manipuler cette conjonction.

```
LET 'fst_aux' "{rel|A->*}[l:List]
               (fst (rel_tl_prop rel l) (rel_list rel l))"
              "{rel|A->*}[l:List]
               (rel_aux rel l)->(rel_tl_prop rel l)";;

LET 'snd_aux' "{rel|A->*}[l:List]
               (snd (rel_tl_prop rel l) (rel_list rel l))"
              "{rel|A->*}[l:List]
               (rel_aux rel l)->(rel_list rel l)";;
```

```
LET 'pair_aux' "{rel|A->*}[l:List]
          [h1:(rel_tl_prop rel l)][h2:(rel_list rel l)]
          <(rel_tl_prop rel l),(rel_list rel l)>(h1,h2)"
         "{rel|A->*}[l:List]
          (rel_tl_prop rel l)->(rel_list rel l)->(rel_aux rel l)";;
```

On montre la stabilité de rel_aux :

```
let REL_CONS_AUX = "{rel|A->*}[c:A][m:List]
          [h1:(rel c)][h2:(rel_aux rel m)]
          (pair_aux rel (Cons c m)
          (tl_tot_co A c m ([m':List](rel_list rel m'))
          (snd_aux rel m h2))
          (rel_list_cons rel c m h1 (snd_aux rel m h2)))";;
LET 'rel_cons_aux' REL_CONS_AUX "{rel|A->*}
          (rel_rec rel (rel_aux rel))";;
```

D'où la propriété cherchée :

```
let REL_TL = "{rel|A->*}[b:A][l:List][h:(rel_list rel (Cons b l))]
        (sym List l (tl_tot A (Cons b l)) (tl_tot_co A b l)
        ([m:List](rel_list rel m))
        (fst_aux rel (Cons b l) (h (rel_aux rel)
                    (pair_aux rel Nil (rel_list_nil rel)
                              (rel_list_nil rel))
                    (rel_cons_aux rel))))";;

LET 'rel_tl' REL_TL "{rel|A->*}[b:A][l:List]
          (rel_list rel (Cons b l))->(rel_list rel l)";;
```

On en déduit que si append l m vérifie rel alors il en est de même de l et de m.

```
PROP 'app_rel_prop1' "{rel|A->*}[n:List][m:List]
          (rel_list rel (Append m n))->(rel_list rel m)";;

let APP_REL1 = "{rel|A->*}[a:A][h0:(rel a)][n:List][m:List]
        (PList m (app_rel_prop1 rel n)
        ([h1:(rel_list rel n)] (rel_list_nil rel))
        ([b:A][m':List][h1':(app_rel_prop1 rel n m')]
        [h2:(rel_list rel (Append (Cons b m') n))]
        (rel_list_cons rel b m' (rel_hd rel a h0 b (Append m' n) h2)
        (h1' (rel_tl rel b (Append m' n) h2)))))";;
LET 'app_rel1' APP_REL1 "{rel|A->*}[a:A](rel a)->(app_rel_prop1 rel)";;

PROP 'app_rel_prop2' "{rel|A->*}[n:List][m:List]
          (rel_list rel (Append m n))->(rel_list rel n)";;

let APP_REL2 = "{rel|A->*}[n:List][m:List]
        (PList m (app_rel_prop2 rel n) <(rel_list rel n)>Id
        ([b:A][m1:List][h1:(app_rel_prop2 rel n m1)]
        [h2:(rel_list rel (Append (Cons b m1) n))]
        (h1 (rel_tl rel b (Append m1 n) h2))))";;
LET 'app_rel2' APP_REL2 "app_rel_prop2";;
```

Réciproquement si m et n vérifient rel alors append m n vérifie rel. Ce résultat se montre par une "rel"-récurrence sur m.

```
let REL_LIST_APP= "{rel|A->*}[m:List][n:List]
            [h1:(rel_list rel m)][h2:(rel_list rel n)]
            (h1 ([m':List](rel_list rel (Append m' n))) h2
            ([b:A][m':List][h3:(rel b)]
             [h4:(rel_list rel (Append m' n))]
            (rel_list_cons rel b (Append m' n) h3 h4)))";;
LET 'rel_list_app' REL_LIST_APP "{rel|A->*}[m:List][n:List]
      (rel_list rel m)->(rel_list rel n)->(rel_list rel (Append m n))";;
```

Une conséquence est que si append m1 m2 et b vérifient rel, il en est de même pour append m1 b.m2.

```
let REL_LIST_MIL = "{rel|A->*}[a:A][h0:(rel a)][b:A][m1:List][m2:List]
            [h1:(rel_list rel (Append m1 m2))][h2:(rel b)]
            (rel_list_app rel m1 (Cons b m2)
            (app_rel1 rel a h0 m2 m1 h1)
            (rel_list_cons rel b m2 h2
            (app_rel2 rel m2 m1 h1)))";;

LET 'rel_list_mil' REL_LIST_MIL "{rel|A->*}[a:A][h0:(rel a)]
                        [b:A][m1:List][m2:List]
   (rel_list rel (Append m1 m2))->(rel b)->(rel_list rel (Mil m1 b m2))";;
```

Toutes ces étapes pour le résultat suivant : si l et m permutent et si l vérifie rel alors il en est de même de m :

```
PROP 'perm_rel_prop' "{rel|A->*}[l:List][m:List]
            (rel_list rel l)->(rel_list rel m)";;

let PERM_REL= "{rel|A->*}[a:A][h0:(rel a)][l:List][m:List]
         [h:(permut l m)]
         (h (perm_rel_prop rel) <(rel_list rel Nil)>Id
         ([b:A][l':List][m1:List][m2:List]
         [h1:(perm_rel_prop rel l' (Append m1 m2))]
         [h2:(rel_list rel (Cons b l'))]
         (rel_list_mil rel a h0 b m1 m2 (h1 (rel_tl rel b l' h2))
         (rel_hd rel a h0 b l' h2)))
         ([l1:List][l2:List][l3:List][t1:(perm_rel_prop rel l1 l2)]
         [t2:(perm_rel_prop rel l2 l3)][t3.(rel_list rel l1)]
         (t2 (t1 t3))))";;

LET 'perm_rel' PERM_REL "{rel|A->*}[a:A][h0:(rel a)][l:List][m:List]
         (permut l m)->(rel_list rel l)->(rel_list rel m)";;
```

Cas particuliers intéressants : rel est inf ou sup :

```
LET 'perm_inf' "[a:A](perm_rel (inf a) a (refl_inf a))"
      "[a:A][l:List][m:List](permut l m)->(inf_list a l)->(inf_list a m)";;

LET 'perm_sup' "[a:A](perm_rel (sup a) a (refl_inf a))"
      "[a:A][l:List][m:List](permut l m)->(sup_list a l)->(sup_list a m)";;
```

Autres propriétés du prédicat permut. le type donne une indication claire de la propriété prouvée :

```
let PERM_APPL = "[l:List][m:List][n:List][h1:(permut l m)]
            (PList n ([n':List](permut (Append n' l) (Append n' m)))
             h1
            ([a:A][n':List][h2:(permut (Append n' l) (Append n' m))]
            (perm_cons a (Append n' l) Nil (Append n' m) h2)))";;

LET 'perm_app_l' PERM_APPL "[l:List][m:List][n:List]
                (permut l m) ->(permut (Append n l) (Append n m))";;

LET 'perm_rè "[l:List](PList l ([m:List](permut m m)) perm_nil
          ([a:A][m:List][h:(permut m m)](perm_cons a m Nil m h)))"
         "[l:List](permut l l)";;

let PERM_APPR = "[l:List][m:List][n:List][h:(permut l m)]
            (h ([l':List][m':List](permut (Append l' n) (Append m' n)))
            (perm_re n)
            ([a:A][l':List][m1:List][m2:List]
            [h1:(permut (Append l' n) (Append (Append m1 m2) n))]
            (Assoc_app m1 (Cons a m2) n
            ([t:List](permut (Append (Cons a l') n) t))
            (perm_cons a (Append l' n) m1 (Append m2 n)
            (sym List (Append m1 (Append m2 n)) (Append (Append m1 m2) n)
            (Assoc_app m1 m2 n) ([t:List](permut (Append l' n) t)) h1))))
            ([l1:List][l2:List][l3:List]
            (perm_tran (Append l1 n) (Append l2 n) (Append l3 n))))";;

LET 'perm_app_r' PERM_APPR "[l:List][m:List][n:List]
                (permut l m)->(permut (Append l n) (Append m n))";;

LET 'perm_app' "[l1:List][m1:List][l2:List][m2:List]
          [h1:(permut l1 m1)][h2:(permut l2 m2)]
          (perm_tran (Append l1 l2) (Append m1 l2) (Append m1 m2)
          (perm_app_r l1 m1 l2 h1) (perm_app_l l2 m2 m1 h2))"
         "[l1:List][m1:List][l2:List][m2:List]
  (permut l1 m1)->(permut l2 m2)->(permut (Append l1 l2) (Append m1 m2))";;
```

Deux listes qui permutent ont même longueur :

```
PROP 'eg_Lgth' "[l:List][m:List]<Nat>(Lgth l)=(Lgth m)";;

let PERM_LGTH = "[l:List][m:List][h:(permut l m)]
            (h eg_Lgth (re Nat 0)
            ([a:A][l':List][m1:List][m2:List]
            [h1:(eg_Lgth l' (Append m1 m2))]
            (Lgth_App m1 (Cons a m2) ([n:Nat]<Nat>(Lgth (Cons a l'))=n)
            (add_n_Sm (Lgth m1) (Lgth m2)
            ([n:Nat]<Nat>(Lgth (Cons a l'))=n)
            (sym Nat (add (Lgth m1) (Lgth m2)) (Lgth (Append m1 m2))
            (Lgth_App m1 m2) ([n:Nat]<Nat>(Lgth (Cons a l'))=(S n))
            (h1 ([n:Nat]<Nat>(Lgth (Cons a l'))=(S n))
            (re Nat (S (Lgth l'))))))))
            ([l1:List][l2:List][l3:List]
            (tran Nat (Lgth l1) (Lgth l2) (Lgth l3))))";;
LET 'perm_Lgth' PERM_LGTH "[l:List][m:List](permut l m)->(eg_Lgth l m)";;
```

Quelques conséquences :

```
let PERM_APP_LT1 = "[l:List][m1:List][m2:List][h:(permut l (Append m1 m2)]
                    (sym Nat (Lgth l) (Lgth (Append m1 m2))
                    (perm_Lgth l (Append m1 m2) h )([n:Nat](LE (Lgth m1) n))
                    (Lgth_App m1 m2 ([n:Nat](LE (Lgth m1) n))
                    (sym Nat (Lgth m1) (add (Lgth m1) 0) (add_n_0 (Lgth m1))
                    ([n:Nat](LE n (add (Lgth m1)(Lgth m2))))
                    (stab_add_l 0 (Lgth m2) (Lgth m1)(peano (Lgth m2))))))";;

LET 'perm_App_LT1' PERM_APP_LT1 "[l:List][m1:List][m2:List]
                    (permut l (Append m1 m2))->(LE (Lgth m1)(Lgth l))";;

let PERM_APP_LT2 = "[l:List][m1:List][m2:List][h:(permut l (Append m1 m2))]
                    (sym Nat (Lgth l) (Lgth (Append m1 m2))
                    (perm_Lgth l (Append m1 m2) h) ([n:Nat](LE (Lgth m2) n))
                    (Lgth_App m1 m2 ([n:Nat](LE (Lgth m2) n))
                    (stab_add_r 0 (Lgth m1) (Lgth m2) (peano (Lgth m1)))))";;

LET 'perm_App_LT2' PERM_APP_LT2 "[l:List][m1:List][m2:List]
                    (permut l (Append m1 m2))->(LE (Lgth m2) (Lgth l))";;
```

On construit maintenant le programme de décomposition. Ce programme travaille sur deux listes qu'il construit au fur et à mesure. La première contient les éléments plus petits que a et la seconde ceux qui sont plus grands. On commence par définir des abbréviations.

```
LET 'INF' "(fst List List)" "List&List->List";;
LET 'SUP' "(snd List List)" "List&List->List";;
LET 'INF_SUP' "[l1:List][l2:List]<List,List>(l1,l2)" "List->List->List&List";;

PROP 'sep_cond' "[a:A][t1:List][t2:List](sup_list a t1)&(inf_list a t2)";;

LET 'sep_pair' "[a:A][m1:List][m2:List]
          [h1:(sup_list a m1)][h2:(inf_list a m2)]
          <(sup_list a m1),(inf_list a m2)>(h1,h2)"
         "[a:A][m1:List][m2:List]
          (sup_list a m1)->(inf_list a m2)->(sep_cond a m1 m2)";;

PROP 'sep_prop' "[a:A][l:List][t1:List][t2:List]
          (sep_cond a t1 t2)&(permut l (Append t1 t2))";;

PROP 'sep_aux' "[a:A][l:List][t:List&List](sep_prop a l (INF t) (SUP t))";;

LET 'sep_aux1' "[a:A][l:List][t:List&List]
          (fst (sep_cond a (INF t) (SUP t))
             (permut l (Append (INF t) (SUP t))))"
         "[a:A][l:List][t:List&List]
          (sep_aux a l t)->(sep_cond a (INF t) (SUP t))";;

LET 'sep_aux_sup' "[a:A][l:List][t:List&List][h:(sep_aux a l t)]
     (fst (sup_list a (INF t)) (inf_list a (SUP t))(sep_aux1 a l t h))"
          "[a:A][l:List][t:List&List]
           (sep_aux a l t)->(sup_list a (INF t))";;
```

```
LET 'sep_aux_inf' "[a:A][l:List][t:List&List][h:(sep_aux a l t)]
    (snd (sup_list a (INF t)) (inf_list a (SUP t)) (sep_aux1 a l t h))"
        "[a:A][l:List][t:List&List]
         (sep_aux a l t)->(inf_list a (SUP t))";;

LET 'sep_aux_perm' "[a:A][l:List][t:List&List]
          (snd (sep_cond a (INF t) (SUP t))
             (permut l (Append (INF t) (SUP t))))"
        "[a:A][l:List][t:List&List]
          (sep_aux a l t)->(permut l (Append (INF t) (SUP t)))";;
```

La spécification du sous-programme :

```
PROP 'separ_spec' "[a:A][l:List]<List&List>Sig((sep_aux a l))";;
```

et son constructeur :

```
let SEPAR_EX = "[a:A][l:List][m:List][n:List][g:(sup_list a m)]
          [h:(inf_list a n)][t:(permut l (Append m n))]
          (exist List&List (sep_aux a l) (INF_SUP m n)
  <(sep_cond a m n),(permut l (Append m n))>((sep_pair a m n g h),t))";;

LET 'separ_ex' SEPAR_EX "[a:A][l:List][m1:List][m2:List]
                [h1:(sup_list a m1)][h2:(inf_list a m2)]
                (permut l (Append m1 m2))->(separ_spec a l)";;
```

On prouve la spécification pour nil

```
LET 'separ_nil' "[a:A](separ_ex a Nil Nil Nil
                (sup_list_nil a) (inf_list_nil a) perm_nil)"
        "[a:A](separ_spec a Nil)";;
```

Passage de l à b.l :

```
let SEPAR_CONS = "[a:A][b:A][l:List][m1:List][m2:List][h1:(sup_list a m1)]
          [h2:(inf_list a m2)][h3:(permut l (Append m1 m2))]
          (tot a b (separ_spec a (Cons b l))
          ([t1:(inf a b)](separ_ex a (Cons b l) m1 (Cons b m2)
          h1 (inf_list_cons a b m2 t1 h2)
          (perm_cons b l m1 m2 h3)))
          ([t2:(inf b a)](separ_ex a (Cons b l) (Cons b m1) m2
          (sup_list_cons a b m1 t2 h1) h2
          (perm_cons b l Nil (Append m1 m2) h3))))";;

LET 'separ_cons' SEPAR_CONS "[a:A][b:A][l:List][m1:List][m2:List]
                [h1:(sup_list a m1)][h2:(inf_list a m2)]
              (permut l (Append m1 m2))->(separ_spec a (Cons b l))";;
```

Le programme général :

```
let SEPAR_ALG = "[a:A][l:List](PList l (separ_spec a) (separ_nil a)
          ([b:A][m:List][h:(separ_spec a m)]
          (h (separ_spec a (Cons b m))
          ([t:List&List][h1:(sep_aux a m t)]
          (separ_cons a b m (INF t) (SUP t)
          (sep_aux_sup a m t h1)(sep_aux_inf a m t h1)
          (sep_aux_perm a m t h1))))))";;

LET 'separ_alg' SEPAR_ALG "[a:A][l:List](separ_spec a l)";;
```

On peut maintenant prouver Quicksort :

Tout d'abord deux abréviations :

```
LET 'sort_tri' "[l:List][m:List](fst (tri m) (permut l m))"
        "[l:List][m:List](sort l m)->(tri m)";;

LET 'sort_perm' "[l:List][m:List](snd (tri m) (permut l m))"
        "[l:List][m:List](sort l m)->(permut l m)";;
```

Preuve de la stabilité noethérienne de la spécification pour une liste non vide :

```
let NOETH_CONS = "[a:A][m:List]
        [h1:[n:List](LT_Lgth n (Cons a m))->(spec_tri n)]

        (separ_alg a m (spec_tri (Cons a m))
        ([t:List&List][h2:(sep_aux a m t)]
        (h1 (INF t)
        (S_LEnm (Lgth (INF t)) (Lgth m)
        (perm_App_LT1 m (INF t) (SUP t) (sep_aux_perm a m t h2)))
        (spec_tri (Cons a m))

        ([m1:List][f1:(tri m1)&(permut (INF t) m1)]
        (h1 (SUP t)
        (S_LEnm (Lgth (SUP t)) (Lgth m)
        (perm_App_LT2 m (INF t) (SUP t) (sep_aux_perm a m t h2)))
        (spec_tri (Cons a m))

        ([m2:List][f2:(tri m2)&(permut (SUP t) m2)]
        (sort_ex (Cons a m) (Mil m1 a m2)
        (tri_app m1 m2 a (sort_tri (INF t) m1 f1)
                (sort_tri (SUP t) m2 f2)

        (perm_sup a (INF t) m1 (sort_perm (INF t) m1 f1)
        (sep_aux_sup a m t h2))
        (perm_inf a (SUP t) m2 (sort_perm (SUP t) m2 f2)
        (sep_aux_inf a m t h2)))
        (perm_cons a m m1 m2
        (perm_tran m (Append (INF t) (SUP t)) (Append m1 m2)
        (sep_aux_perm a m t h2)
        (perm_app (INF t) m1 (SUP t) m2 (sort_perm (INF t) m1 f1)
        (sort_perm (SUP t) m2 f2))))))))))";;

LET 'noeth_cons' NOETH_CONS "[a:A][m:List](noeth_ind spec_tri (Cons a ))";;
```

Et enfin le programme :

```
let QUICK_PROOF = "[l:List]
    (List_noeth l spec_tri ([m:List]
    (Listp m (noeth_ind spec_tri m)
        ([t1:(Nullp m)]
        (t1 (noeth_ind spec_tri)
        ([h1:[n:List](LT_Lgth n Nil)->(spec_tri n)]sort_nil)))
        ([t2:(Consp m)](t2 (noeth_ind spec_tri)
        ([a:A][n:List](noeth_cons a n)))))))";;
```

```
LET 'quick_proof' QUICK_PROOF "spec_tri";;
```

5.3. Remarques

Notons que dans cet exemple comme pour celui du programme de mise en page, la difficulté provient des définitions. On peut contester le choix de celles-ci. Par exemple le choix des listes triées peut semblée trop orientée vers l'application à Quicksort. Une définition plus simple (orientée en fait vers le tri par insertion) est :

[l:List]{P|List->}*
(P Nil)->([a:A][m:List](P m)->(inf_list a m)->(P (Cons a m)))->(P l)

On appelle *tri_ins* cette définition. On vérifie aisément que pour toute liste l on a : *(tri l)->(ins_tri l)* Pour cela il nous faut trouver des preuves de *(ins_tri Nil)* et de

[m1:List][m2:List][a:A]
(ins_tri m1)->(ins_tri m2)->(sup_list a m1)->(inf_list a m2)
->(ins_tri (Mil m1 a m2))

Il suffit ensuite d'appliquer la preuve de (*tri l*) à ces résultats. Le premier est une application directe de la définition de *ins_tri*, le second se montre par récurrence sur $m1$. Le cas $m1$=*Nil* est aussi une application directe de la définition de *tri_ins*, à savoir : *[a:A][m:List](ins_tri m)->(inf_list a m)->(ins_tri (Cons a m))*
Le seul point plus délicat est de montrer que, étant donnés b de type A et $n1$ une liste, si h est une preuve de

[m2:List][a:A](ins_tri n1)->(ins_tri m2)->(sup_list a n1)->(inf_list a m2)
->(ins_tri (Mil n1 a m2))

alors soit :

[n2:List][c:A][h1:(ins_tri (Cons b n1))][h2:(ins_tri n2)]
[h3:(sup_list c (Cons b n1))][h4:(inf_list c n2)]

il faut construire un terme de type : *(ins_tri (Mil (Cons b n1) c n2))* , c'est-à-dire à β-conversion près, *(ins_tri (Cons b (Mil n1 c n2)))*.
Pour cela il suffit d'avoir *(ins_tri (Mil n1 c n2))* et *(inf_list b (Mil n1 c n2))*.
(ins_tri (Mil n1 c n2)) s'obtient par application de l'hypothèse de récurrence h à: $n2$, c, une preuve de (*ins_tri* $n1$), $h2$, une preuve de *sup_list* c $n1$, et $h4$. Les preuves de (*ins_tri* $n1$) et (*sup_list* c $n1$) sont des conséquences de $h1$ et $h3$, leur déduction est analogue à la preuve de : *(rel_list (Cons a m))->(rel_list m)* développée dans le programme. Reste à prouver (*inf_list b* (*Mil* $n1$ c $n2$)). C'est un peu lourd à écrire, il faut montrer que *(ins_tri (Cons b n1))->(inf_list b n1)*. D'autre part $h3$ donne (*sup_listcn* 1), on a donc en résumé :
$b \leq n1 \leq c \leq n2$, on en déduit aisément (mais pas très succintement) que $b \leq$ (*Mil* $n1$ a $n2$).
Notons que cette preuve n'intervient pas dans l'algorithme. On peut la voir comme une "meta" preuve de l'équivalence de deux définitions.

On peut discuter aussi de la définition du prédicat *permut*. On ne dispose pas ici d'une fonction simple à calculer caractérisant l'équivalence (comme *normal* pour la mise en page), on a donc employé une définition constructive. Dans cette définition on introduit la propriété de transitivité (qui ne semble

pas démontrable sinon). Ceci nous oblige, à chaque démonstration de *(permut l m)→(Q l m)*, à montrer en plus la transitivité de *Q*. Il semble à postériori qu'une autre définition convient, à savoir la stabilité des listes par toute relation ne portant que sur ces éléments. Formellement:

*permbis = [l:List][m:List]{rel|List-> *}(rel_list rel l)->(rel_list rel m)*

Prouver *(permut l m)->(inf_list a l)->(inf_list a m)* serait alors trivial ainsi que la transitivité et la réflexivité de *permbis*. Par contre montrer que :
(permut l (Append m1 m2))->(permut (Cons a l) (Mil m1 a m2))
demande un peu plus de travail.

Une dernière remarque sur l'utilisation d'un principe de récurrence noethérienne sur les listes: si on l'écrit de manière plus générale pour un type quelconque et une relation noethérienne sur ce type, on voit que c'est simplement une preuve de terminaison d'un programme récursif tel qu'il y ait décroissance sur toute branche de l'arbre des appels récursifs.

6. Conclusion

L'une des premières choses surprenantes est la relative longueur dans le calcul des constructions de petites preuves arithmétiques triviales comparativement à d'autres raisonnement plus complexes qui, eux s'expriment très simplement dans ce langage. Ceci est dû au caractère très primitif des constructions. En fait on peut y remédier simplement en ayant à sa disposition une bibliothèque de preuves assez riche. C'est ce que l'on a commencé à faire en appendice. On peut aussi espérer synthétiser automatiquement les preuves de lemmes simples. Combiné avec une implémentation du système plus interactive, cela doit permettre une utilisation plus agréable du système.

Une autre difficulté (qui n'est pas étrangère à la première) est de savoir ce qu'il est nécessaire de prendre comme axiome. Les axiomes de constructions des entiers, des listes ... sont dus au fait que l'on utilise ces structures de manière générale pour construire d'autres objets. Mais quand on cherche à montrer les propriétés de ces objets, ces types et la β-réduction sont souvent insuffisants, il faut faire intervenir la manière dont a été construite la preuve qui dépend beaucoup de la structure de l'élément de départ (entier, liste ..). Ces axiomes sont vrais pour des termes clos, et il est également clair que lorsque l'on applique un programme, par exemple à un entier particulier, celui-ci est de la forme (S (S ..S(0)..). La difficulté vient de l'introduction d'hypothèses dans l'environnement. Un bon exemple de ces phénomènes est l'utilisation du prédicat de décidabilité de la relation *eg* dans la définition du nombre de voix.

Notons aussi que pour les entiers, les listes ou les arbres ces axiomes sont aussi les preuves de terminaison des constructions par récurrence.

Les constructions apparaissent comme un langage qui permet à la fois des codages de preuves, de programmes, de types de données concrets comme les entiers, les listes ou les arbres, de types de données plus complexes limités à un certain nombre d'opérations de constructions (entiers pairs, arbre de stockage, liste triée..) ou de gestion d'erreurs. Toutes ces notions sont introduites ici dans des cas particuliers mais un certain nombre de constructions doivent pouvoir être systématisées offrant ainsi des primitives plus proches des outils classiques de programmation.

Nous espérons avoir montré par ces exemples les grandes possibilités du calcul des constructions comme langage de programmation de développement d'algorithmes. En effet même s'il faut un petit peu de pratique pour le manipuler, l'utilisation de constantes le rend très lisible. Les algorithmes "portent" leur preuve avec eux, ce qui peut aider à modifier des programmes après coup sans risque d'introduire des erreurs. Il semble donc que ce langage soit un bon outil pour une programmation rigoureuse.

7. Bibliographie

[1] R.Backhouse. *Algorithm development in Martin-Löf's type theory*, University of Essex (Juillet 84).

[2] J.L.Bates et R.L.Constable *Proofs as Programs* Dept of Computer Science, Cornell University (Février 83)

[3] R.Boyer etJ.Moore, *MJRTY, a fast majority vote algorithm*, University of Texas at Austin (Février 81).

[4] N.G. De Bruijn, *A survey of the project Automath*, Curry Volume, Academic Press (1980).

[5] Th.Coquand, *Une théorie des constructions*, Thèse de troisième cycle, Université Paris VII (Janvier 85)

[6] Th.Coquand, *An Analysis of Girard's Paradox* proposé au Colloque in Logic on Computer Science (Juin 86).

[7] Th.Coquand and G.Huet, *A theory of constructions*, Preliminary version, presenté à: the international Symposium on semantics and data types. Sophia Antipolis (Juin 84)

[8] Th.Coquand and G.Huet, *Constructions : A higher order proof system for mechanizing mathematics*, EUROCAL 85,Linz,Austria (Avril 85).

[9] Th.Coquand and G.Huet, *A calculus of constructions*

[10] Th.Coquand and G.Huet, *Concepts Mathématiques et Informatiques Formalisés dans le calcul des constructions* Logic Colloquium, Orsay, July 85.

[11] J.Y. Girard, *Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types*, Proceedings of the Second Scandinavian Logic Symposium,Ed. J.E. Fenstad, North Holland,pp 63-92 (1970).

[12] W.A. Howard, *The formulae-as-types Notion of Construction*, Curry volume, Academic Press (1980).

[13] P. Martin-Löf, *Constructive Mathematics and Computer Programming*, Logic,Methodology and Philosophy of Science VI,pp. 153-175, North-Holland (1980)

[14] P. Martin-Löf, *Intuitionistic Type Theory*, Studies in Proof Theory, Bibliopolis (1984).

[15] R.P.Nederpelt, *An approach to theorem proving on the basis of a typed lambda calculus*, Lectures Notes in Computer Science 87 : 5th CConference on Automated Deduction,Les Arcs,France,Springer-Verlag (1980).

[16] B.Nordström, *Programming in Constructive Set Theory : Some Examples*, Proceedings of the conference on Fonctional Programming Languages and Computer Architecture, Portmouth, New Hampshire, p.141-154 (octobre 1981)

[17] J.Smith, *The identification of propositions and types in Martin-Löf's type theory : a programming examples*, University of Göteborg Sweden (Novembre 82)

8. Appendice

On présente ici les constructions de base, c'est-à-dire que l'on commence avec une théorie vide. D'un point de vue implémentation, nous travaillons sur l'interprèteur ML dans un environnement où l'on a accès à un parser, un pretty-printer et un type-checker.

Dans ce qui suit, les commentaires sont en italiques.

CONNECTEURS LOGIQUE :
Implication, conjonction, disjonction, quantificateurs, absurde, négation.

Implication intuitionniste :

Définition :

```
let ARROW = "!A,B.A->B";;
PROP '->' ARROW;;
LET_SYNTAX '->' ['(';'->';')'];;
```

Preuve de A->A :

```
let ID = "!A.[u:A]u";;
LET 'Id' ID "!A.A->A";;
LET_SYNTAX 'Id' ['<';'>Id'];;
```

Produit et conjonction :

Définition :

```
let PROD = "!A,B,C.(A->B->C)->C";;
PROP '&' PROD;;
LET_SYNTAX '&' ['(';'&';')'];;
```

Formation de la paire :

```
let PAIR = "!A,B.[x:A][y:B]!C.[z:A->B->C](z x y)";;
LET '<>' PAIR "!A,B.A->B->A&B";;
LET_SYNTAX '<>' ['<';',';'>(';',';')'];;
```

Première projection :

```
let FST = "!A,B.[x:A&B](x A [y:A][z:B]y)";;
LET 'fst' FST "!A,B.A&B->A";;
LET_SYNTAX 'fst' ['<';',';'>fst(';')'];;
```

Seconde projection :

```
let SND = "!A,B.[x:A&B](x B [y:A][z:B]z)";;
LET 'snd' SND "!A,B.A&B->B";;
LET_SYNTAX 'snd' ['<';',';'>snd(';')'];;
```

Somme disjointe :

Définition :

```
let SUM = "!A,B,C.(A->C)->(B->C)->C";;
PROP '+' SUM;;
LET_SYNTAX '+' ['(';'+';')'];;
```

Injection à gauche :

```
let INL = "!A,B.[x:A]!C.[y:A->C][z:B->C](y x)";;
LET 'Inl' INL "!A,B.A->A+B";;
```

Injection à droite :

```
let INR = "!A,B.[x:B]!C.[y:A->C][z:B->C](z x)";;
LET 'Inr' INR "!A,B.B->A+B";;
```

Quantificateurs :

Quantificateur universel :

```
let PI = "!A.{P|A->*}[x:A](P x)";;
PROP 'Pi' PI;;
LET_SYNTAX 'Pi' ['<';'>Pi(';')'];;
```

Quantificateur existentiel :

```
let SIGMA = "!A.{P|A->*}!B.([x:A](P x)->B)->B";;
PROP 'Sig' SIGMA;;
LET_SYNTAX 'Sig' ['<';'>Sig(';')'];;
```

Introduction du quantificateur :

```
let Sig_intro = "!A.{P|A->*}[x:A](P x)-><A>Sig(P)";;
let EXIST = "!A.{P|A->*}[x:A][y:(P x)]!B.[f:[x:A](P x)->B](f x y)";;
LET 'exist' EXIST Sig_intro;;
```

Négation

Proposition absurde (ne possède pas de preuve) :

```
let VOID = "!A.A";;
PROP '{}' VOID;;
```

De {} on déduit toute proposition.

```
let CONTR = "!A.[u:{}](u A)";;
LET 'contr' CONTR "!A.{}->A";;
```

Négation :

```
let NOT = "!A.A->{}";;
PROP '~' NOT;;
LET_SYNTAX '~' ['~';''];;
```

PROPRIETES DES RELATIONS BINAIRES, EGALITE :

Réflexivité, transitivité et symétrie :

```
let REFL = "!A.{R|A->A->*}[x:A](R x x)";;
PROP 'refl' REFL;;
let TRANS = "!A.{R|A->A->*}[x:A][y:A][z:A](R x y)->(R y z)->(R x z)";;
PROP 'trans' TRANS;;
let SYME = "!A.{R|A->A->*}[x:A][y:A](R x y)->(R y x)";;
PROP 'symè SYME;;
```

Egalité :

```
let EQUAL = "!A.[n:A][m:A]{P|[x:A]*}(P n)->(P m)";;
PROP '=' EQUAL;;
LET_SYNTAX '=' ['<';'>';'=';''];;
```

```
let RE = "!A.[n:A]{P|[u:A]*}[h:(P n)]h";;
LET 'rè RE "!A.[x:A]<A>x=x";;
let SYM = "!A.[n:A][m:A][h:<A>n=m]
        (h ([u:A]<A>u=n) (re A n))";;
LET 'sym' SYM "!A.[x:A][y:A](<A>x=y)-><A>y=x";;
let TRAN = "!A.[x:A][y:A][z:A][h1:<A>x=y][h2:<A>y=z]
        (h2 ([u:A]<A>x=u) h1)";;
LET 'tran' TRAN "!A.[x:A][y:A][z:A](<A>x=y)->(<A>y=z)-><A>x=z";;
```

BOOLEENS

Définition

```
let BOOL="!A.A->A->A";;
PROP 'bool' BOOL;;
```

Les deux preuves de bool :

```
let TRUE= "!A.[u:A][v:A]u";;
LET 'truè TRUE "bool";;

let FALSE = "!A.[u:A][v:A]v";;
LET 'falsè FALSE "bool";;
```

Branchement conditionnel

```
let IF = "!A.[b:bool](b A)";;
LET 'if' IF "!A.bool->A->A->A";;
```

Conjonction de deux booléens

```
let ET = "[a:bool][b:bool](a bool b false)";;
LET 'et' ET "bool->bool->bool";;
```

SOMMES DISJOINTES CONSTRUITES

On axiomatise la propriété de construction des sommes disjointes :

```
PROP 'C_sum' "!A,B.[z:A+B]{P|(A+B)->*}
        ([x:A](P (Inl A B x)))->([y:B](P (Inr A B y)))->(P z)";;

AXIOM 'axi_sum' "C_sum";;
```

Soit B un type et f1,f2 deux éléments de B, on suppose que l'on construit un élément de type B par cas :

```
LET 'func_cas' "!A1,A2.[u:A1+A2]!B.[f1:B][f2:B](u B ([a:A1]f1) ([b:A2]f2))"
        "!A1,A2.[u:A1+A2]!B.[f1:B][f2:B]B";;
```

On montre que dans chacun des cas on a bien construit l'élément voulu. Il faut remarquer que l'on a utilisé dans la définition de fun_cas une preuve arbitraire de A1+A2. Or on a juste l'information suivante :
(fun_cas A1 A2 (Inl A1 A2 a1) B f1 f2) $\rightarrow_\beta$ f1
et (fun_cas A1 A2 (Inr A1 A2 a2) B f1 f2) $\rightarrow_\beta$ f2
d'où la nécessité de l'axiome de construction et de l'exclusion mutuelle de A1 et A2.

```
PROP 'cas1_prop' "!A1,A2,B.[f1:B][f2:B][u:A1+A2]
        <B>f1=(func_cas A1 A2 u B f1 f2)";;
```

```
PROP 'cas2_prop' "!A1,A2,B.[f1:B][f2:B][u:A1+A2]
                  <B>f2=(func_cas A1 A2 u B f1 f2)";;

let CASE1 = "!A1,A2.[u:A1+A2][h:A1->A2->{}]!B.[f1:B][f2:B][g:A1]
          (axi_sum A1 A2 u (cas1_prop A1 A2 B f1 f2)
          ([b:A1](re B f1))
          ([a:A2](h g a (cas1_prop A1 A2 B f1 f2 (Inr A1 A2 a)))))";;

LET 'case1' CASE1 "!A1,A2.[u:A1+A2][h:A1->A2->{}]!B.[f1:B][f2:B]
                   A1->(cas1_prop A1 A2 B f1 f2 u)";;

let CASE2 = "!A1,A2.[u:A1+A2][h:A1->A2->{}]!B.[f1:B][f2:B][g:A2]
          (axi_sum A1 A2 u (cas2_prop A1 A2 B f1 f2)
          ([b:A1](h b g (cas2_prop A1 A2 B f1 f2 (Inl A1 A2 b))))
          ([a:A2](re B f2)))";;

LET 'case2' CASE2 "!A1,A2.[u:A1+A2][h:A1->A2->{}]!B.[f1:B][f2:B]
                   A2->(cas2_prop A1 A2 B f1 f2 u)";;
```

ENTIERS

Définition des entiers : itérateur polymorphe à partir d'un élement.

```
let ent = "!A.[f:A->A][a:A]A";;
PROP 'Nat' ent;;

let zero = "!A.[f:A->A][a:A]a";;
LET '0' zero "Nat";;
```

Fonction successeur sur les entiers :

```
let succ = "[n:Nat]!A.[f:A->A][a:A](f(n A f a))";;
LET 'S' succ "Nat->Nat";;
```

Autre preuve de Nat :

```
LET '1' "(S 0)" "Nat";;
```

Itération :
Soit A de type *, f de type Nat&A->A, a de type A et n un entier (iter A f a n) est un objet de type A qui peut être vu comme le résultat z du programme suivant :
z:=a;for i:=0 to n-1 do z:=f(i,z)

```
let ITER = "!A.[f:Nat&A->A][a:A]
         [n:Nat]
         let it_pair = (n Nat&A
              ([st:Nat&A]<Nat,A>((S <Nat,A>fst(st)),(f st)))
              <Nat,A>(0,a)) in
         <Nat,A>snd(it_pair)";;
LET 'iter' ITER "!A.(Nat&A->A)->A->Nat->A";;
```

On définit le prédecesseur par itération :

```
let PRED = "[n:Nat](iter Nat ([st:Nat&Nat]<Nat,Nat>fst(st)) 0 n)";;
LET 'pred' PRED "Nat->Nat";;
```

Relations d'ordres sur les entiers.

Propriété de stabilité croissante :

```
let HER = "{P|[u:Nat]*}[n:Nat](P n)->(P (S n))";;
PROP 'her' HER;;
```

Propriété de stabilité décroissante :

```
let HIN ="{P|[u:Nat]*}[n:Nat](P (S n))->(P n)";;
PROP 'hin' HIN;;
```

Inférieur ou égal ≤ :

```
let infl = "[n:Nat][m:Nat]{P|[u:Nat]*}(P n)->(her P)->(P m)";;
PROP 'LE' infl;;
```

Supérieur ou égal ≥ :

```
let supl = "[n:Nat][m:Nat]{P|[u:Nat]*}(P n)->(hin P)->(P m)";;
PROP 'GE' supl;;
```

Le principe de récurrence sur les entiers s'écrit 0≤n, avoir une preuve de (LE 0 n) c'est savoir écrire n comme S(S...(S 0)..).

```
let construit = "[n:Nat]{P|[u:Nat]*}(P 0)->(her P)->(P n)";;
PROP 'C' construit;;
```

Inégalite stricte, n<m se définit comme S(n)≤m :

```
let infs = "[n:Nat][m:Nat]{P|[u:Nat]*}(P(S n))->(her P)->(P m)";;
PROP 'LT' infs;;
```

Preuve de la réflexivité de ≤ et ≥ :

```
let LE_n_n = "[n:Nat]{P|[u:Nat]*}[h1:(P n)][h2:(her P)]h1";;
LET 'LE_n_n' LE_n_n "(refl Nat LE)";;

let GE_n_n = "[n:Nat]{P|[u:Nat]*}[h1:(P n)][h2:(hin P)]h1";;
LET 'GE_n_n' GE_n_n "(refl Nat GE)";;
```

Preuve de la transitivité de ≤ et ≥ :

```
let TRANS_LE = "[n:Nat][m:Nat][p:Nat][h1:(LE n m)][h2:(LE m p)]
          {P|[u:Nat]*}[h3:(P n)][h4:(her P)]
          (h2 P (h1 P h3 h4) h4)";;
LET 'trans_LE' TRANS_LE "(trans Nat LE)";;

let TRANS_GE = "[n:Nat][m:Nat][p:Nat][h1:(GE n m)][h2:(GE m p)]
          {P|[u:Nat]*}[h3:(P n)][h4:(hin P)]
          (h2 P (h1 P h3 h4) h4)";;
LET 'trans_GE' TRANS_GE "(trans Nat GE)";;
```

Preuve de n≤S(n) :

```
let LE_N_SN = "[n:Nat]{P|Nat->*}[h1:(P n)][h2:(her P)](h2 n h1)";;
LET 'LE_n_Sn' LE_N_SN "[n:Nat](LE n (S n))";;
```

Preuve de S(n)≥n :

```
let GE_SN_N = "[n:Nat]{P|Nat->*}[h1:(P (S n))][h2:(hin P)](h2 n h1)";;
LET 'GE_Sn_n' GE_SN_N "[n:Nat](GE (S n) n)";;
```

Preuve de n≤m ⇒ m≥n :

```
let LE_GE = "[n:Nat][m:Nat][h:(LE n m)]
        (h ([u:Nat](GE u n)) (GE_n_n n)
        ([u:Nat][h:(GE u n)](trans_GE (S u) u n (GE_Sn_n u) h)))";;
LET 'LE_GE' LE_GE "[n:Nat][m:Nat](LE n m)->(GE m n)";;
```

Preuve de $n \geq m \Rightarrow m \leq n$:

```
let GE_LE = "[n:Nat][m:Nat][h:(GE n m)]
        (h ([u:Nat](LE u n)) (LE_n_n n)
        ([u:Nat][h:(LE (S u) n)](trans_LE u (S u) n (LE_n_Sn u) h)))";;
LET 'GE_LE' GE_LE "[n:Nat][m:Nat](GE n m)->(LE m n)";;
```

Preuve de $n \leq m \Rightarrow S(n) \leq S(m)$:

```
let LE_S = "[n:Nat][m:Nat][h1:(LE n m)]{P|[u:Nat]*}
                [h2:(P (S n))][h3:(her P)]
          (h1 ([u:Nat](P (S u))) h2 ([u:Nat][h4:(P (S u))]
                        (h3 (S u) h4)))";;
LET 'S_LEnm' LE_S "[n:Nat][m:Nat](LE n m)->(LE (S n) (S m))";;
```

Preuve de $n \leq m \Rightarrow m = n$ ou $n < m$

```
let EG_LT = "[n:Nat][m:Nat][h1:(LE n m)]
        (h1 ([u:Nat] ((<Nat>n=u)+(LT n u)))
        (Inl <Nat>n=n (LT n n) (re Nat n))
        ([u:Nat][h2:((<Nat>n=u)+(LT n u))]
        (h2 ((<Nat>n=(S u))+(LT n (S u)))
        ([x1:<Nat>n=u](Inr <Nat>n=(S u) (LT n (S u))
                (x1 ([x:Nat](LT n (S x))) (LE_n_n (S n)))))
        ([x2:(LT n u)](Inr <Nat>n=(S u) (LT n (S u))
                (trans_LE (S n) u (S u) x2 (LE_n_Sn u)))))))";;
LET 'LE_EGLT' EG_LT "[n:Nat][m:Nat](LE n m)->((<Nat>n=m)+(LT n m))";;
```

Opérations sur les entiers :
(on remarque l'utilisation des entiers comme itérateurs.)

```
let ADD = "[n:Nat](n Nat S)";;
LET 'add' ADD "Nat->Nat->Nat";;

let MULT2 = "[n:Nat](add n n)";;
LET 'mult2' MULT2 "Nat->Nat";;
let MULT = "[n:Nat][m:Nat](n Nat (add m) 0)";;
LET 'mult' MULT "Nat->Nat->Nat";;

let EXP = "[n:Nat][m:Nat](m Nat (mult n) (S 0))";;
LET 'exp' EXP "Nat->Nat->Nat";;
```

Preuve de : $a \leq b \Rightarrow \forall c \in N \cdot\ a+c \leq b+c$.

```
let STAB_ADD_PROP = "[a:Nat][b:Nat][c:Nat]
        (LE a b)->(LE (add a c) (add b c))";;
```

```
let STAB_ADD = "[a:Nat][b:Nat][c:Nat][h:(LE a b)]
          (h ([n:Nat](LE (add a c) (add n c)))
          (LE_n_n (add a c))
          ([n:Nat][f:(LE (add a c)(add n c))]
          (trans_LE (add a c) (add n c) (add (S n) c)
                f (LE_n_Sn (add n c)))))";;
LET 'stab_add_r' STAB_ADD STAB_ADD_PROP;;
```

PROPRIETES ARITHMETIQUES SUR LES ENTIERS CONSTRUITS.

On montre ici des résultats sur les entiers qui nécessitent l'utilisation de l'axiome de peano exprimant le principe de récurrence sur les entiers.

```
AXIOM 'peano' "[n:Nat](C n)";;
```

La première conséquence est n≥0, ce qui peut s'exprimer comme un principe de récurrence descendante.

```
let REC_INV_PROP = "[n:Nat]{P|Nat->*}(P n)->(hin P)->(P 0)";;
PROP 'rec_inv_prop' REC_INV_PROP;;

let REC_INV = "[n:Nat](LE_GE 0 n (peano n))";;
LET 'rec_inv' REC_INV "rec_inv_prop";;
```

preuve de n≤m ou m<n :

```
let LE_OU_LT = "[n:Nat][m:Nat]
          let inj1 = [u:Nat](Inl (LE u m) (LT m u)) in
          let inj2 = [u:Nat](Inr (LE u m) (LT m u)) in
          (peano n ([u:Nat](LE u m)+(LT m u))
          (inj1 0 (peano m))
          ([u:Nat][hyp_rec:(LE u m)+(LT m u)]
          (hyp_rec (LE (S u) m)+(LT m (S u))
          ([t1:(LE u m)] (LE_EGLT u m t1 (LE (S u) m)+(LT m (S u))
                ([t11:<Nat>u=m]
             (inj2 (S u) (t11 ([v:Nat](LT v (S u))) (LE_n_n (S u)))))
                ([t12:(LT u m)](inj1 (S u) t12))))
          ([t2:(LT m u)](inj2 (S u)
                (trans_LE (S m) u (S u) t2 (LE_n_Sn u)))))))";;
LET 'LE_ou_LT' LE_OU_LT "[n:Nat][m:Nat]((LE n m)+(LT m n))";;
```

Comme application on peut construire la fonction max.

```
let MAX = "[n:Nat][m:Nat](LE_ou_LT n m Nat ([h1:(LE n m)] m)
                        ([h2:(LT m n)] n))";;
LET 'max' MAX "Nat->Nat->Nat";;
```

Par β-réduction 0+n donne n et S(n)+m donne S(n+m). Par contre les preuves de n+0=n et n+S(m)=S(n+m) ne s'obtiennent que sur les entiers que l'on sait construire.

n+0=n.

```
let ADD_n_0 = "[n:Nat](peano n ([v:Nat]<Nat>v=(add v 0))
          (re Nat 0)
          ([v:Nat][h:<Nat>v=(add v 0)]
          (h ([w:Nat]<Nat>(S v)=(S w)) (re Nat (S v)))))";;
LET 'add_n_0' ADD_n_0 "[n:Nat]<Nat>n=(add n 0)";;
```

n+S(m)=S(n+m).

```
PROP 'ADD_S' "[n:Nat][m:Nat]<Nat>(S (add n m))=(add n (S m))";;
let ADD_N_SM = "[n:Nat][m:Nat](peano n ([u:Nat](ADD_S u m))
                  (re Nat (S m))
                  ([u:Nat][h:(ADD_S u m)]
                  (h ([v:Nat]<Nat>(S (add (S u) m))=(S v))
                  (re Nat (S (S (add u m)))))))";;
LET 'add_n_Sm' ADD_N_SM "ADD_S";;
```

Une application : 2S(n) = S(S(2n))

```
let MULT2_S = "[n:Nat](add_n_Sm (S n) n)";;
LET 'mult2_S' MULT2_S "[n:Nat]<Nat>(S (S (mult2 n)))=(mult2 (S n))";;
```

Symétrie de l'addition :

```
let SYM_ADD = "[n:Nat][m:Nat](peano n
        ([u:Nat]<Nat>(add u m)=(add m u)) (add_n_0 m)
        ([u:Nat][h:<Nat>(add u m)=(add m u)]
        (add_n_Sm m u ([w:Nat]<Nat>(add (S u) m)=w)
        (h ([z:Nat]<Nat>(add (S u) m)=(S z))
        (re Nat (S (add u m)))))))";;
LET 'sym_add' SYM_ADD "[n:Nat][m:Nat]<Nat>(add n m)=(add m n)";;
```

On en déduit a≤b ⇒ a+c≤b+c.

```
let STAB_SYM ="[a:Nat][b:Nat][c:Nat][h:(LE a b)]
        (sym_add a c ([u:Nat](LE u (add c b)))
        (sym_add b c ([v:Nat](LE (add a c) v))(stab_add_r a b c h)))";;
LET 'stab_add_l' STAB_SYM "[a:Nat][b:Nat][c:Nat]
                (LE a b)->(LE (add c a)(add c b))";;
```

Propriétés de la fonction prédécesseur.

pred(S (n)) = n.

```
let PRED_SN_N = "[n:Nat]
        let pred_S_rec = [m:Nat][h:<Nat>(pred (S m))=m]
          (h ([u:Nat]<Nat>(pred (S (S m)))=(S u))
          (re Nat (S (pred (S m))))) in
        (peano n ([u:Nat]<Nat>(pred (S u))=u)
            (re Nat 0) pred_S_rec)";;
LET 'pred_Sn_n' PRED_SN_N "[n:Nat]<Nat>(pred (S n))=n";;
```

Version symétrique :

```
let N_PRED_SN = "[n:Nat](sym Nat (pred (S n)) n (pred_Sn_n n))";;
LET 'n_pred_Sn' N_PRED_SN "[n:Nat]<Nat>n=(pred (S n))";;
```

pred(n)≤n (pas d'inégalité stricte car pred(0)=0).

```
let LE_PREDN_N = "[n:Nat](peano n ([u:Nat](LE (pred u) u))
                    (LE_n_n 0)
                    ([u:Nat][h:(LE (pred u) u)]
                    (n_pred_Sn u ([v:Nat](LE v (S u))) (LE_n_Sn u))))";;
LET 'LE_predn_n' LE_PREDN_N "[n:Nat](LE (pred n) n)";;
```

Preuve de : $n \leq S(pred(n))$.

```
let LE_N_SPREDN = "[n:Nat](peano n ([v:Nat](LE v (S (pred v))))
                (LE_n_Sn 0)
                ([v:Nat][h:(LE v (S (pred v)))]
                (n_pred_Sn v ([w:Nat](LE (S v) (S w)))
                (LE_n_n (S v)))))";;
LET 'LE_n_Spredn' LE_N_SPREDN "[n:Nat](LE n (S (pred n)))";;
```

Preuve de : $n \leq m \Rightarrow pred(n) \leq pred(m)$.

```
let LE_PRED = "[n:Nat][m:Nat][h:(LE n m)]
            (h ([u:Nat](LE (pred n) (pred u)))
            (LE_n_n (pred n))
            ([u:Nat][f:(LE (pred n) (pred u))]
            (trans_LE (pred n) (pred u) (pred (S u)) f
            (n_pred_Sn u ([v:Nat](LE (pred u) v)) (LE_predn_n u)))))";;
LET 'LE_pred' LE_PRED "[n:Nat][m:Nat](LE n m)->(LE (pred n) (pred m))";;
```

Conséquence : $S(n) \leq S(m) \Rightarrow n \leq m$.

```
let S_LE = "[n:Nat][m:Nat][h:(LE (S n) (S m))]
         (pred_Sn_n n ([v:Nat](LE v m))
         (pred_Sn_n m ([w:Nat](LE (pred (S n)) w))
         (LE_pred (S n) (S m) h)))";;
LET 'S_LE' S_LE "[n:Nat][m:Nat](LE (S n) (S m))->(LE n m)";;
```

Plus généralement : $a+b \leq a+c \Rightarrow b \leq c$.

```
PROP 'SIMPL_ADD_PROP' "[a:Nat][b:Nat][c:Nat]
                (LE (add a b) (add a c))->(LE b c)";;

let SIMPL_ADD_L = "[a:Nat][b:Nat][c:Nat]
                (peano a ([u:Nat](SIMPL_ADD_PROP u b c))
                <(LE b c)>Id
                ([u:Nat][f:(SIMPL_ADD_PROP u b c)]
                [g:(LE (add (S u) b)(add (S u) c))]
                (f (S_LE (add u b) (add u c) g))))";;
LET 'simpl_add_l' SIMPL_ADD_L "SIMPL_ADD_PROP";;
```

La symétrie de l'addition donne : $b+a \leq c+a \Rightarrow b \leq c$.

```
let SIMPL_ADD_R = "[a:Nat][b:Nat][c:Nat][h:(LE (add b a) (add c a))]
                (simpl_add_l a b c
                (sym_add b a ([w:Nat](LE w (add a c)))
                (sym_add c a ([w:Nat](LE (add b a) w)) h)))";;
LET 'simpl_add_r' SIMPL_ADD_R "[a:Nat][b:Nat][c:Nat]
                        (LE (add b a) (add c a))->(LE b c)";;
```

Associativité de l'addition :

```
let ADD_ASSOC = "[a:Nat][b:Nat][c:Nat]
          (peano a ([n:Nat]<Nat>(add n (add b c))=(add (add n b) c))
          (re Nat (add b c))
          ([n:Nat][h:<Nat>(add n (add b c))=(add (add n b) c)]
          (h ([v:Nat]<Nat>(add (S n) (add b c))=(S v))
          (re Nat (S (add n (add b c)))))))";;
LET 'add_assoc_r' ADD_ASSOC "[a:Nat][b:Nat][c:Nat]
                    <Nat>(add a (add b c))=(add (add a b) c)";;

let ADD_ASS_SYM = "[a:Nat][b:Nat][c:Nat]
          (sym Nat (add a (add b c))(add (add a b) c)
          (add_assoc_r a b c))";;
LET 'add_assoc_l' ADD_ASS_SYM "[a:Nat][b:Nat][c:Nat]
                    <Nat>(add (add a b) c)=(add a (add b c))";;
```

Nous utiliserons également un autre axiome de peano sur les entiers à savoir que S(x) est différent de 0.

```
AXIOM 'axi_ari' "[n:Nat]~<Nat>(S n)=0";;
```

LISTES

Définition :

```
let LIST = "!A,B.[f:A->B->B][g:B]B";;
PROP 'list' LIST;;
```

Liste vide :

```
let NIL= "!A,B.[f:A->B->B][g:B]g";;
LET 'nil' NIL "list";;
```

Opérations sur les listes : cons et append.

```
let CONS = "!A.[x:A][l:(list A)]!B.[f:A->B->B][g:B](f x (l B f g))";;
LET 'cons' CONS "!A.A->(list A)->(list A)";;

let APPEND = "!A.[l1:(list A)][l2:(list A)](l1 (list A) (cons A) l2)";;
LET 'append' APPEND "!A.(list A)->(list A)->(list A)";;
```

Longueur d'une liste :

```
let LENGTH = "!A.[l:(list A)](l Nat ([x:A][n:Nat](S n)) 0)";;
LET 'length' LENGTH "!A.(list A)->Nat";;
```

Prédicat de construction sur les listes. Une preuve de (pred_list A l) si l est une liste d'élements de A, est un moyen de construire la liste l à partir de la fonction cons, d'élements de A et de la liste vide.

```
let PRED_LIST = "!A.[l:(list A)]{P|(list A)->*}
     (P (nil A))->([x:A][l':(list A)](P l')->(P (cons A x l')))->(P l)";;
PROP 'pred_list' PRED_LIST;;
```

On suppose les listes construites :

```
AXIOM 'axi_list' "pred_list";;
```

Résultats supplémentaires sur les listes. Tout d'abord les prédicats pour les listes vides ou non.

```
PROP 'nullp' "!A.[l:(list A)]<(list A)>(nil A)=l";;

PROP 'consp' "!A.[l:(list A)]{P|(list A)->*}
        ([a:A][m:(list A)](P (cons A a m)))->(P l)";;
```

Deux résultats évidents : la liste vide est vide et une liste formée d'un cons ne l'est pas :

```
let CAS_NIL = "!A.(re (list A) (nil A))";;
LET 'cas_nil' CAS_NIL "!A.(nullp A (nil A))";;

let CAS_CONS = "!A.[a:A][l:(list A)]{P|(list A)->*}
          [h:[b:A][m:(list A)](P (cons A b m))](h a l)";;
LET 'cas_cons' CAS_CONS "!A.[a:A][l:(list A)](consp A (cons A a l))";;
```

A partir de l'axiome de construction des listes on a une preuve qu'une liste est soit vide soit formée d'un cons :

```
PROP 'listp' "!A.[l:(list A)](nullp A l)+(consp A l)";;

let CASE_LIST = "!A.[l:(list A)](axi_list A l (listp A)
              (Inl (nullp A (nil A)) (consp A (nil A)) (cas_nil A))
              ([a:A][m:(list A)][h:(listp A m)]
              (Inr (nullp A (cons A a m))(consp A (cons A a m))
                 (cas_cons A a m))))";;

LET 'case_list' CASE_LIST "listp";;
```

Une liste vide n'est pas formée d'un cons :

```
let NIL_NO_CONS = "!A.[l:(list A)][h1:(nullp A l)][h2:(consp A l)]
              (h2 ([m:(list A)](nullp A m)->{})
              ([a:A][m:(list A)][h:(nullp A (cons A a m))]
              (axi_ari (length A m)
                 (h ([p:(list A)]<Nat>(length A p)=0)
                   (re Nat 0)))) h1)";;
LET 'nil_no_cons' NIL_NO_CONS "!A.[l:(list A)]
                      (nullp A l)->(consp A l)->{}";;
```

On veut construire les deux fonctions donnant l'élément de tête et le reste d'une liste. Ceci n'est bien défini que pour les listes non vides. Or si on veut montrer par exemple que a.l=b.m => a=b et l=m on s'aperçoit que l'usage des fonctions partielles posent des difficultés. On rend donc les fonctions arbitrairement totales en posant tl_tot nil = nil (comme on a posé pred 0 = 0) pour la recherche de la liste sans son élément de tête. Il est moins naturel de trouver ce qu'est l'élément de tête de la liste vide, on paramètre donc cette fonction par la valeur de hd_tot nil.

```
LET 'hd_tot' "!A.[a0:A][p:(list A)](p A ([c:A][p':A]c) a0)"
        "!A.A->(list A)->A";;
```

La construction de tl_tot nécessite un procédé d'itération sur un couple de listes semblable à celui utilisé pour la fonction prédecesseur . Ceci correspond à une récursion primitive sur les listes : F (a.l) = H (a,l,F(l))

Abréviations pour la manipulation de paires de listes :

```
LET 'fst_lst' "!A.(fst (list A) (list A))" "!A.(list A)&(list A)->(list A)";;
LET 'snd_lst' "!A.(snd (list A) (list A))" "!A.(list A)&(list A)->(list A)";;
LET 'pair_lst' "!A.[l:(list A)][m:(list A)]<(list A),(list A)>(l,m)"
        "!A.(list A)->(list A)->(list A)&(list A)";;

PROP 'eq_lst_lst' "!A.[l:(list A)&(list A)][m:(list A)&(list A)]
            <(list A)&(list A)>l=m";;

let TL_PRIM = "!A.[l:(list A)]
        (l (list A)&(list A)
          ([a:A][g:(list A)&(list A)]
          (pair_lst A (cons A a (fst_lst A g)) (fst_lst A g)))
          (pair_lst A (nil A) (nil A)))";;
LET 'tl_prim' TL_PRIM "!A.(list A)->(list A)&(list A)";;
```

On prouve par récurrence sur la structure de la liste que tl_tot a.l = l, après avoir prouvé le résultat analogue pour tl_prim.

```
PROP 'tl_co_prop' "!A.[m:(list A)][a:A]
    (eq_lst_lst A (pair_lst A (cons A a m) m)(tl_prim A (cons A a m)))";;

let TL_PRIM_CONS = "!A.[l:(list A)]
            (axi_list A l (tl_co_prop A)
            ([a:A](re (list A)&(list A)
            (pair_lst A (cons A a (nil A)) (nil A))))
            ([b:A][m:(list A)][f:(tl_co_prop A m)][a:A]
            (f b ([p:(list A)&(list A)]
            (eq_lst_lst A
            (pair_lst A (cons A a (cons A b m)) (cons A b m))
            (pair_lst A (cons A a (fst_lst A p)) (fst_lst A p))))
            (re (list A)&(list A)
            (pair_lst A (cons A a (cons A b m)) (cons A b m))))))";;

LET 'tl_prim_cons' TL_PRIM_CONS "!A.[l:(list A)](tl_co_prop A l)";;

LET 'tl_tot' "!A.[l:(list A)](snd_lst A (tl_prim A l))"
        "!A.(list A)->(list A)";;

LET 'tl_tot_co' "!A.[a:A][l:(list A)]
            (tl_prim_cons A l a
            ([m:(list A)&(list A)]<(list A)>l=(snd_lst A m))
            (re (list A) l))"

            "!A.[a:A][l:(list A)]
            <(list A)>l=(tl_tot A (cons A a l))";;
```

Quelques résultats supplémentaires.

L'associativité de append.:

```
PROP 'assoc_app_prop' "!A.[l:(list A)][m:(list A)][n:(list A)]
    <(list A)>(append A l (append A m n))=(append A (append A l m) n)";;
```

```
let ASSOC_APP = "!A.[l:(list A)][m:(list A)][n:(list A)]
          (axi_list A l
          ([l':(list A)](assoc_app_prop A l' m n))
          (re (list A) (append A m n))
          ([b:A][l':(list A)][h:(assoc_app_prop A l' m n)]
          (h ([t:(list A)]
        <(list A)>(append A (cons A b l') (append A m n))=(cons A b t))
          (re (list A) (cons A b (append A l' (append A m n)))))))";;

LET 'assoc_app' ASSOC_APP "!A.[l:(list A)][m:(list A)][n:(list A)]
     <(list A)>(append A l (append A m n))=(append A (append A l m) n)";;
```

La longueur de la liste concaténée de deux listes est égale à la somme des longueurs des deux listes :

```
PROP 'lgth_app_prop' "!A.[l:(list A)][m:(list A)]
       <Nat>(add (length A l) (length A m))=(length A (append A l m))";;

let LGTH_APP = "!A.[l:(list A)][m:(list A)](axi_list A l
          ([l':(list A)](lgth_app_prop A l' m))
          (re Nat (length A m))
          ([a:A][l':(list A)][h:(lgth_app_prop A l' m)]
          (h ([n:Nat]
          <Nat>(add (length A (cons A a l')) (length A m))=(S n))
           (re Nat (S (add (length A l') (length A m)))))))";;

LET 'lgth_app' LGTH_APP "!A.[l:(list A)][m:(list A)]
       <Nat>(add (length A l)(length A m))=(length A (append A l m))";;
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique